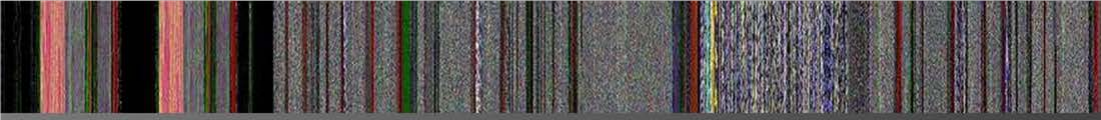


Memory Thinking

C & C++

Linux Diagnostics

Dmitry Vostokov
Software Diagnostics Services

Memory Thinking for C & C++ Linux Diagnostics

Slides with Descriptions Only

Dmitry Vostokov
Software Diagnostics Services

OpenTask

Memory Thinking for C & C++ Linux Diagnostics: Slides with Descriptions Only

Published by OpenTask, Republic of Ireland

OpenTask books are available through booksellers and distributors worldwide. For further information or comments, send requests to press@opentask.com.

A CIP catalog record for this book is available from the British Library.

ISBN-13: 978-1912636570 (Paperback)

Revision 1.00 (December 2023)

Table of Contents

Preface

This full-color reference book is a part of the Accelerated C & C++ for Linux Diagnostics training course organized by Software Diagnostics Services (www.patterndiagnostics.com). The text contains slides, brief notes highlighting particular points, and replicated source code fragments that are easy to copy into your favorite IDE. The book's detailed Table of Contents makes the usual Index redundant. We hope this reference is helpful for the following audiences:

- C and C++ developers who want to deepen their knowledge;
- Software engineers developing and maintaining products on Linux platforms;
- Technical support, escalation, DevSecOps, cloud and site reliability engineers dealing with complex software issues;
- Quality assurance engineers who test software on Linux platforms;
- Security and vulnerability researchers, reverse engineers, malware and memory forensics analysts.

If you encounter any error, please use the contact form on the Software Diagnostics Services web site or, alternatively, via Twitter @DumpAnalysis.

Facebook group:

http://www.facebook.com/groups/dumpanalysis

LinkedIn page and group:

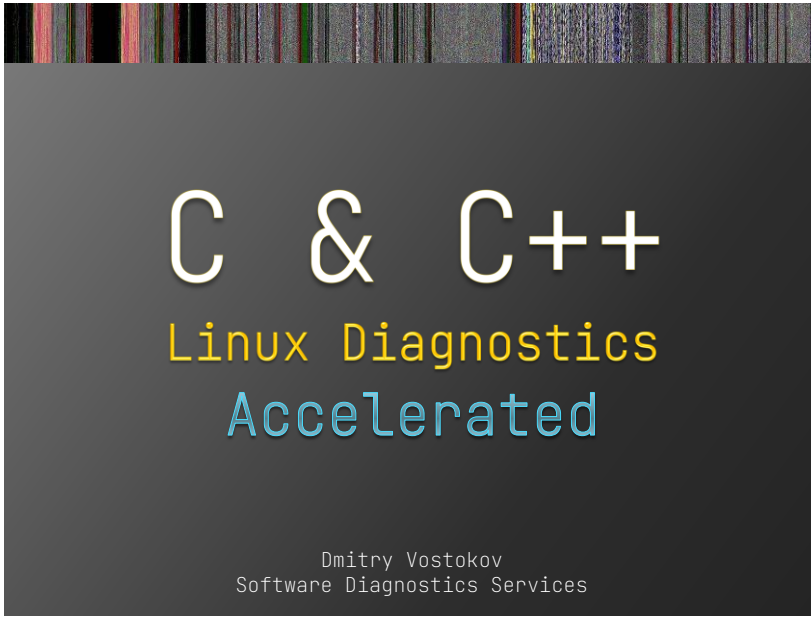
https://www.linkedin.com/company/software-diagnostics-institute/
https://www.linkedin.com/groups/8473045/

About the Author

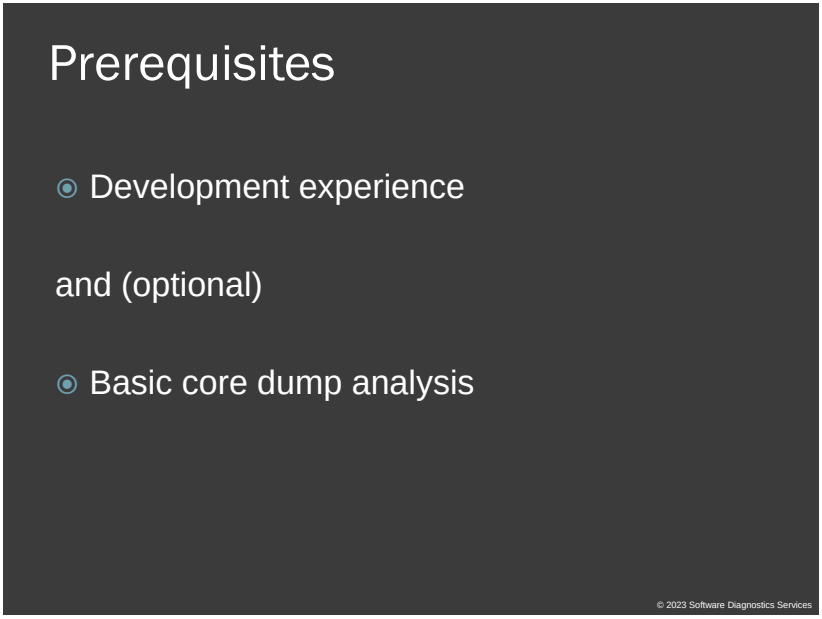
Dmitry Vostokov is an internationally recognized expert, speaker, educator, scientist, inventor, and author. He founded the pattern-oriented software diagnostics, forensics, and prognostics discipline (Systematic Software Diagnostics) and Software Diagnostics Institute (DA+TA: DumpAnalysis.org + TraceAnalysis.org). Vostokov has also authored over 50 books on software diagnostics, anomaly detection and analysis, software and memory forensics, root cause analysis and problem solving, memory dump analysis, debugging, software trace and log analysis, reverse engineering, and malware analysis. He has over 25 years of experience in software architecture, design, development, and maintenance in various industries, including leadership, technical, and people management roles. Dmitry also founded Syndromatix, Anolog.io, BriteTrace, DiaThings, Logtellect, OpenTask Iterative and Incremental Publishing (OpenTask.com), Software Diagnostics Technology and Services (former Memory Dump Analysis Services) PatternDiagnostics.com, and Software Prognostics. In his spare time, he presents various topics on Debugging.TV and explores Software Narratology, its further development as Narratology of Things and Diagnostics of Things (DoT), Software Pathology, and Quantum Software Diagnostics. His current interest areas are theoretical software diagnostics and its mathematical and computer science foundations, application of formal logic, artificial intelligence, machine learning and data mining to diagnostics and anomaly detection, software diagnostics engineering and diagnostics-driven development, diagnostics workflow and interaction. Recent interest areas also include cloud native computing, security, automation, functional programming, applications of category theory to software development and big data, and diagnostics of artificial intelligence.

Introduction

Original Training Course Name

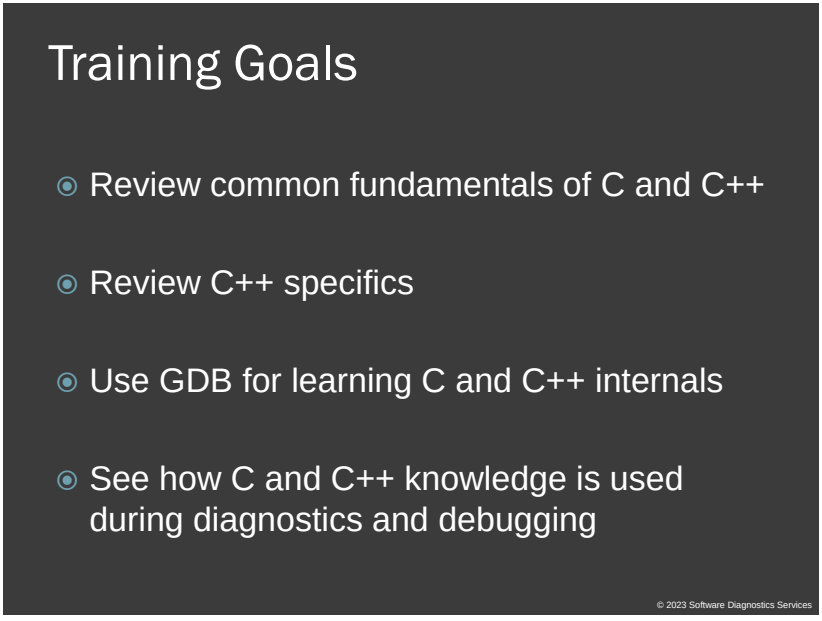


Prerequisites



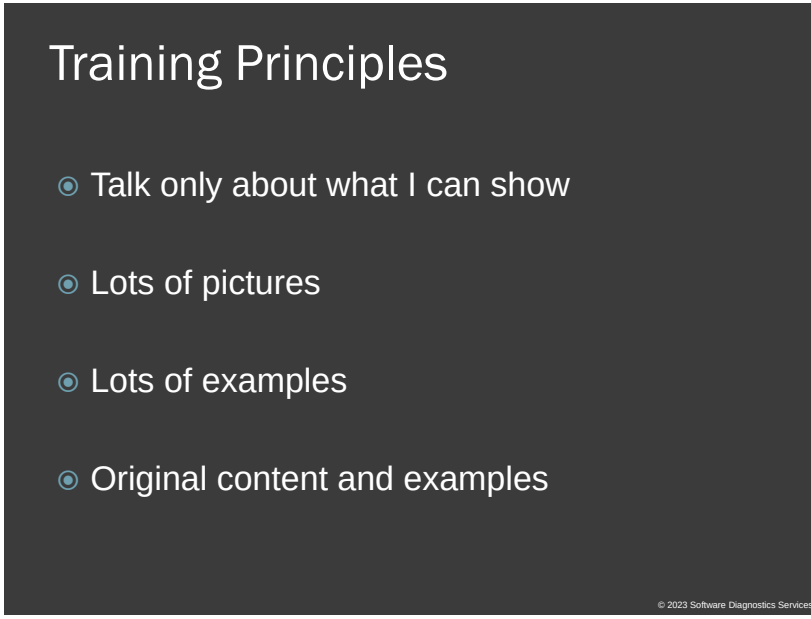
To get most of this training, you are expected to have basic development experience in a programming language other than C or C++ and optional basic memory dump analysis experience. I also included the necessary x64 and A64 disassembly reviews for some topics.

Training Goals



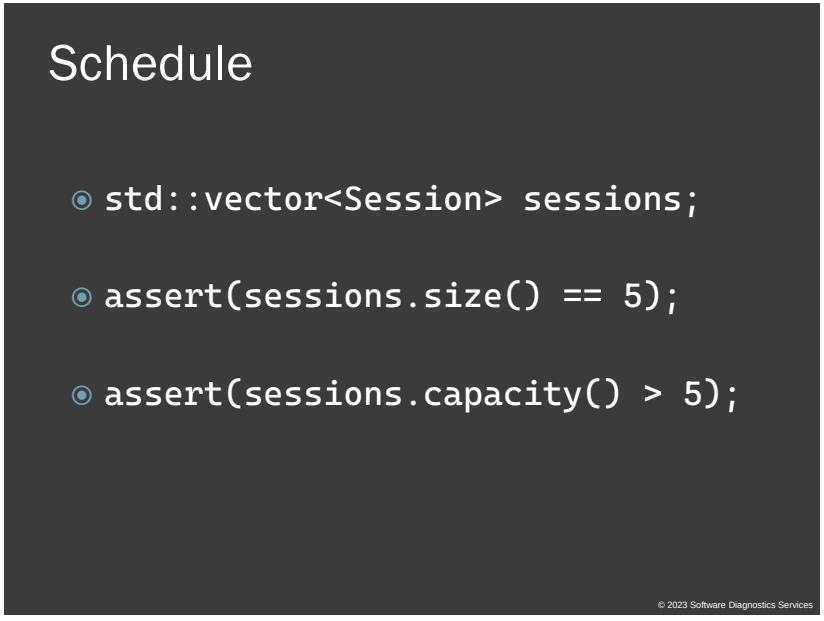
Our primary goal is to learn C and C++ and its internals in an accelerated fashion. First, we review common C and C++ fundamentals necessary for software diagnostics. Then, we learn various C++ features with a focus on memory and internals. We also see examples of how the knowledge of C and C++ helps in diagnostics and debugging.

Training Principles



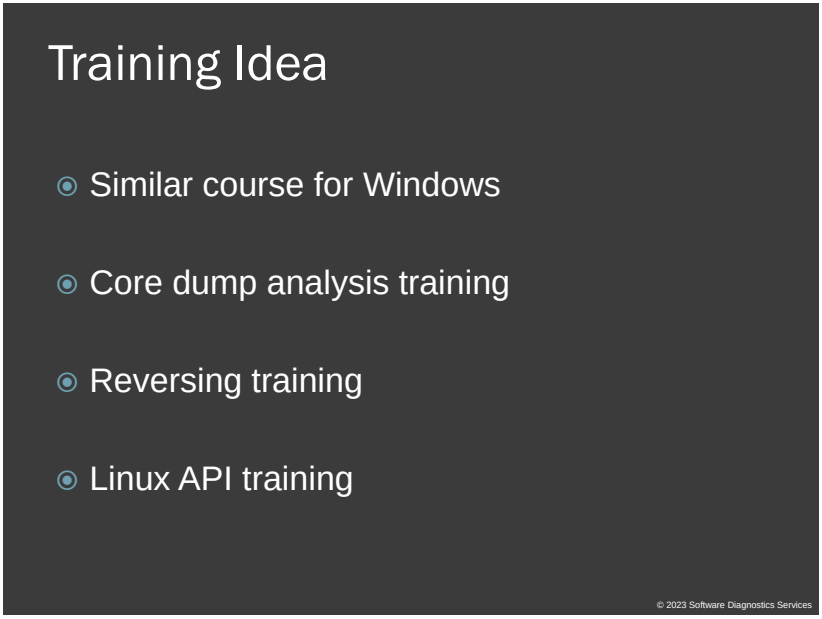
There were many training formats to consider, and I decided that the best way is to concentrate on slides and code examples you can verify.

Schedule



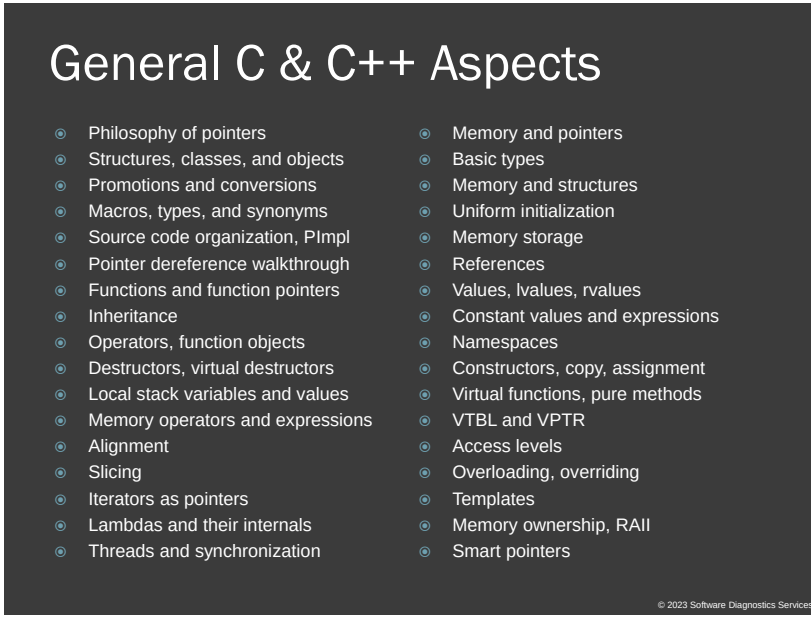
I plan the training to have only 5 two-hour sessions, but I may extend it to more sessions if necessary to fit all necessary material in sufficient detail.

Training Idea



After I created a similar Windows-based training, it was natural to port it to Linux. Also, attendees of core dump analysis and reversing training courses asked questions related to C and C++, and I realized that they would have also benefitted if they had this training. This training may also fill some gaps between these courses. Finally, I recently developed the **Accelerated Linux API** training course (see the References section at the end of the book), where solid knowledge of classic C and C++ is assumed, and the current C and C++ course may provide such knowledge.

General C & C++ Aspects

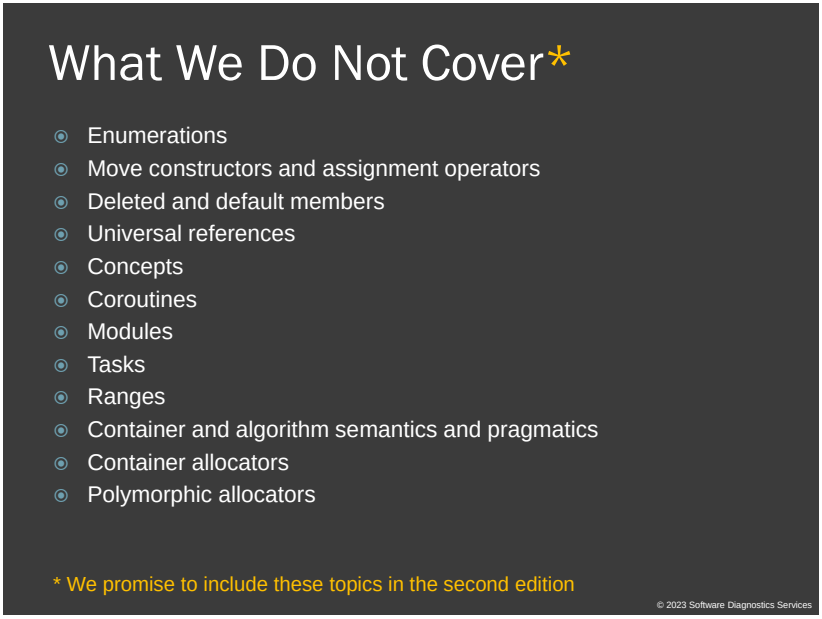


The general C and C++ aspects that we discuss in this course:

- Philosophy of pointers
- Structures, classes, and objects
- Promotions and conversions
- Macros, types, and synonyms
- Source code organization, PImpl
- Pointer dereference walkthrough
- Functions and function pointers
- Inheritance
- Operators, function objects
- Destructors, virtual destructors
- Local stack variables and values
- Memory operators and expressions
- Alignment
- Slicing

- Iterators as pointers
- Lambdas and their internals
- Threads and synchronization
- Memory and pointers
- Basic types
- Memory and structures
- Uniform initialization
- Memory storage
- References
- Values, lvalues, rvalues
- Constant values and expressions
- Namespaces
- Constructors, copy, assignment
- Virtual functions, pure methods
- VTBL and VPTR
- Access levels
- Overloading, overriding
- Templates
- Memory ownership, RAII
- Smart pointers

What We Do Not Cover

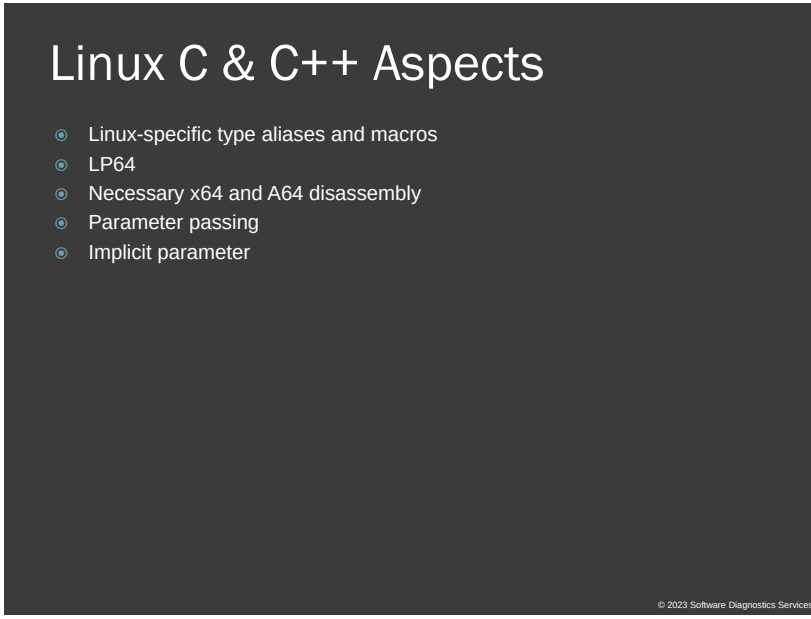


There are some C++ topics that we did not include:

- Enumerations
- Move constructors and assignment operators
- Deleted and default members
- Universal references
- Concepts
- Coroutines
- Modules
- Tasks
- Ranges
- Container and algorithm semantics and pragmatics
- Container allocators
- Polymorphic allocators

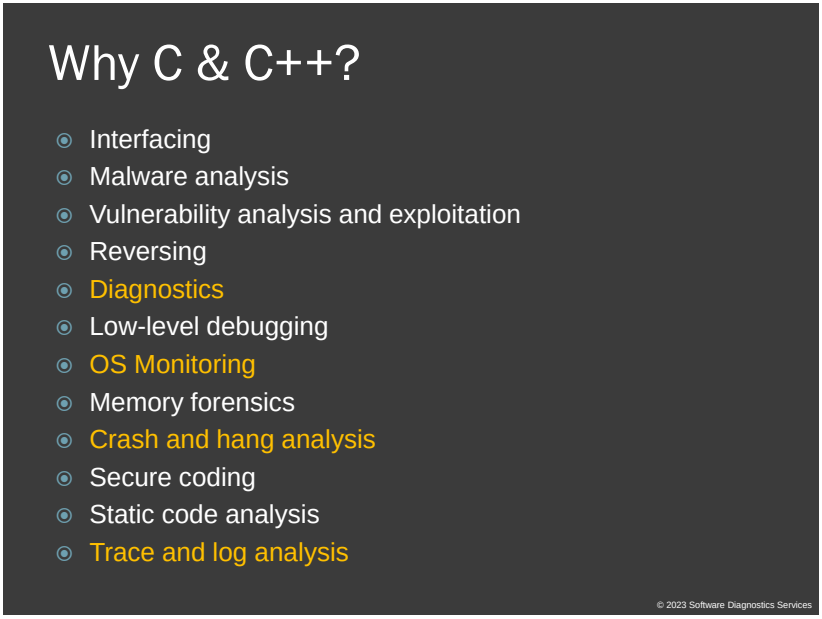
We promise to include these topics in the second edition of this course.

Linux C & C++ Aspects



In addition, we also discuss related Linux aspects, including:

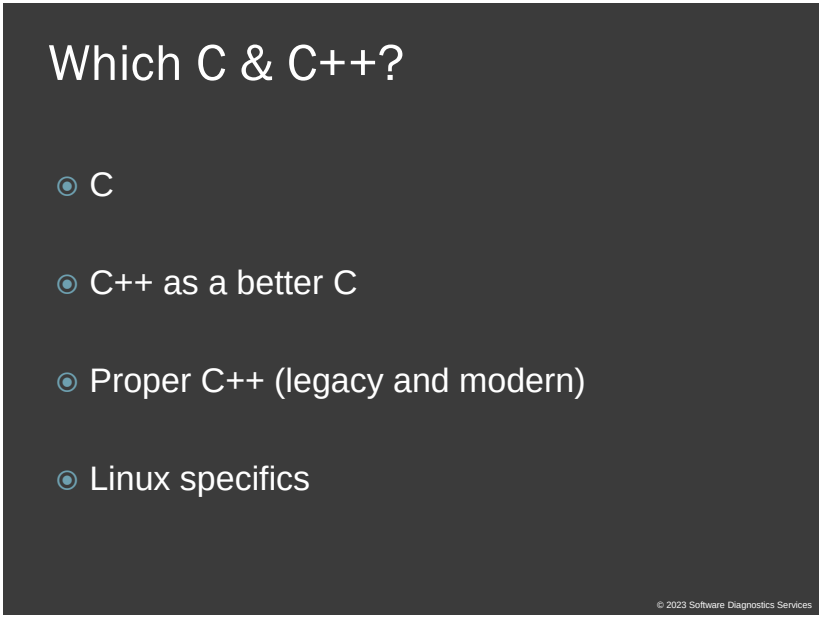
- Linux-specific type aliases and macros
- LP64
- Necessary x64 and A64 disassembly
- Parameter passing
- Implicit parameter



First, why did we create this course? Even if you don't develop in C and C++, the knowledge of C and C++ and their internals is necessary for many software construction and post-construction activities:

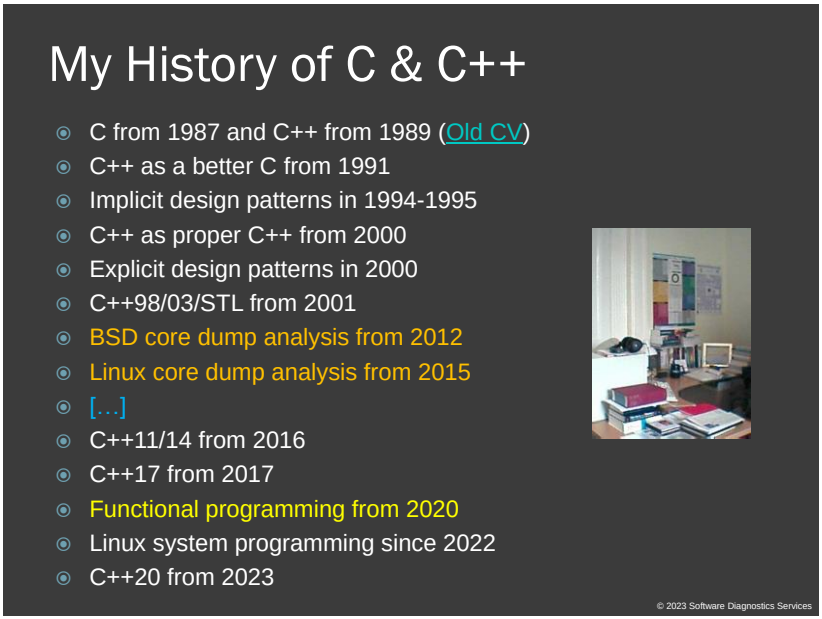
- Interfacing
- Malware analysis
- Vulnerability analysis and exploitation
- Reversing
- **Diagnostics**
- Low-level debugging
- **OS Monitoring**
- Memory forensics
- **Crash and hang analysis**
- Secure coding
- Static code analysis
- **Trace and log analysis**

In this training, we mostly look at C and C++ from a software diagnostics perspective. This perspective includes memory dump analysis and, partially, trace and log analysis. The knowledge of C and C++ is tacitly assumed in my other courses, where most abnormal software behavior modeling exercises are written in C and C++. Of course, there is an intersection of what we learn with other areas.



Which C and C++? We look at a unified presentation approach combining all C and C++ variants. Since this course is about diagnostics and not designing and implementing code, we do not make distinctions. It is not possible to cover all the differences in the short time that we have. We also describe things as they are in Linux programming (and narrowly from a Linux system programming perspective), not as they ought to be from the latest C++ standards.

My History of C & C++



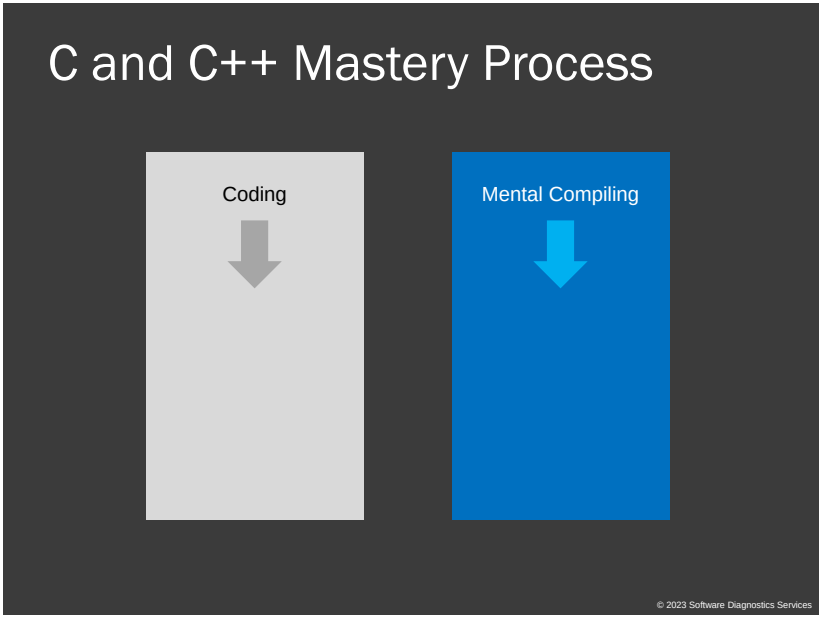
This history slide is only about C and C++ languages. Despite many years, it is still easy to recall when I started learning C. It was shortly after I started my university education. And although my first programming language was FORTRAN, I read the classic K&R book in a library. C++ is harder to recall, but most likely, it was in 1989, at least according to my old CV, which is the source of truth. I definitely started using C++ in commercial projects around 1991 but used it as a better C, and there was no standard template library (STL) at that time. I recall some fascinating C++ GUI frameworks for MS-DOS, like Zinc. In 1994-1995, I designed a word processor, and in the process, I implicitly used many design patterns I later discovered in the GOF book in 2000. The authors also use a word processor for illustration. I mainly understood C++ as C++ in 2000 when I read a book about CORBA distributed object technology that used C++. I continued learning C++ by reading many books of that time and learned the merits of using STL and also how to use it effectively. In 2001, I joined a company that developed C++ static analysis tools, and this greatly improved my C++ knowledge up to the expert level at

that time. C++03 didn't have major changes compared to C++98, and this is why I included it with C++98 for the year 2001. In 2003, things turned out unexpectedly as I moved from full-time development using C++ to full-time memory dump analysis of C++ programs with Mac OS X core dump analysis of user space (BSD) in 2012 and Linux user space core dump analysis in 2015. I continued using C++03 for writing diagnostic tools, though. In 2016, I learned that the language completely changed to C++11/14. I came back to full-time C++ programming in late 2017, where I also started using language features from C++17. In 2020, I moved to functional programming in Scala, which also influenced my C++ coding for new projects. Now, I have started using C++20 - a bit late since C++23 is already available, and I am switching from Scala to Rust.

Zinc
https://en.wikipedia.org/wiki/Zinc_Application_Framework

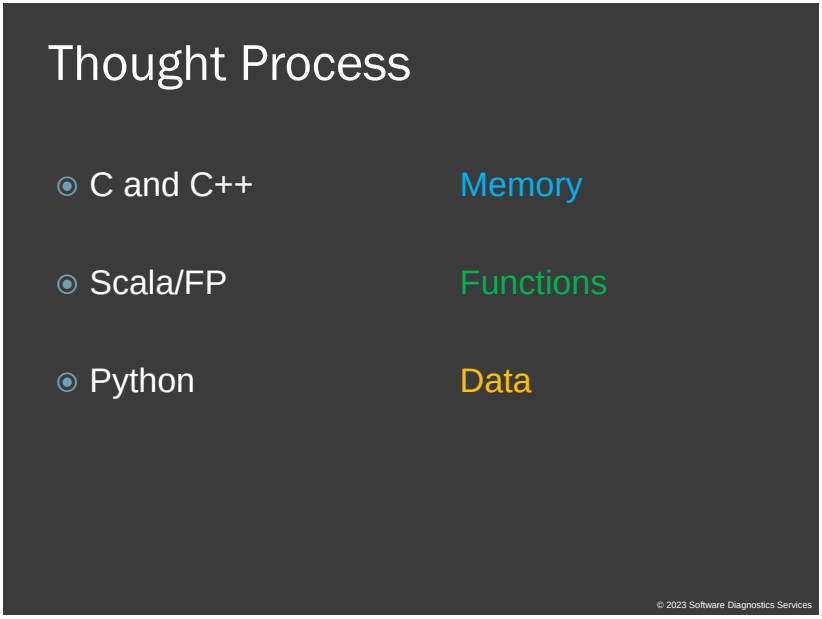
Old CV
https://opentask.com/Vostokov/CV.htm

C and C++ Mastery Process



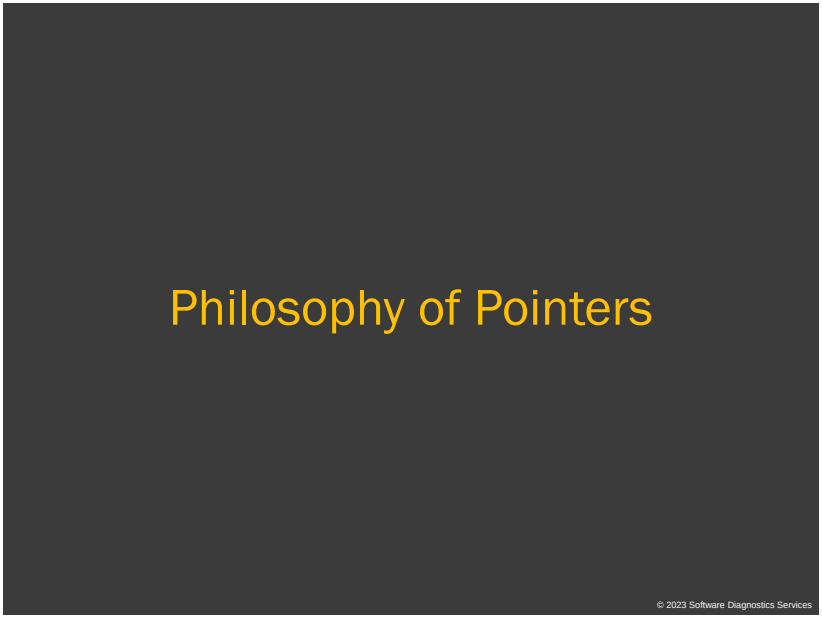
Despite high-level features in C++, there's still much low-level overlap with C, and when I program in C and C++, I mentally compile to memory. This helps when I have a doubt about whether this or that construct is safe. And I also believe that looking at how C and C++ constructs are implemented in memory greatly helps in learning these languages.

Thought Process



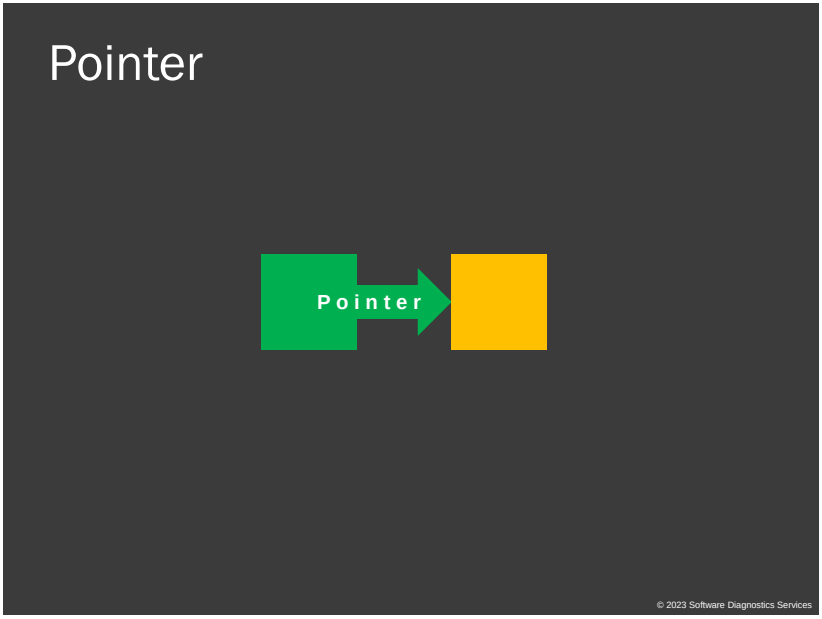
This slide about a thought process when using a programming language is perhaps controversial. With C and C++, we think about memory; with Scala/FP, we think about functions; and with Python, we think about data.

Philosophy of Pointers



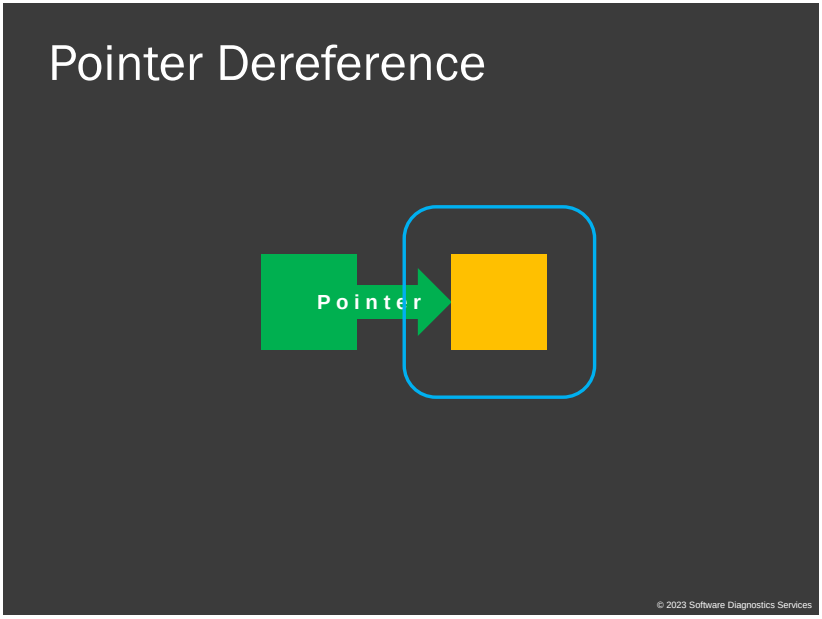
We start with pointers, the most important concept in C and also in C++. I originally created this approach in 2015 but now extended it for this training.

Pointer



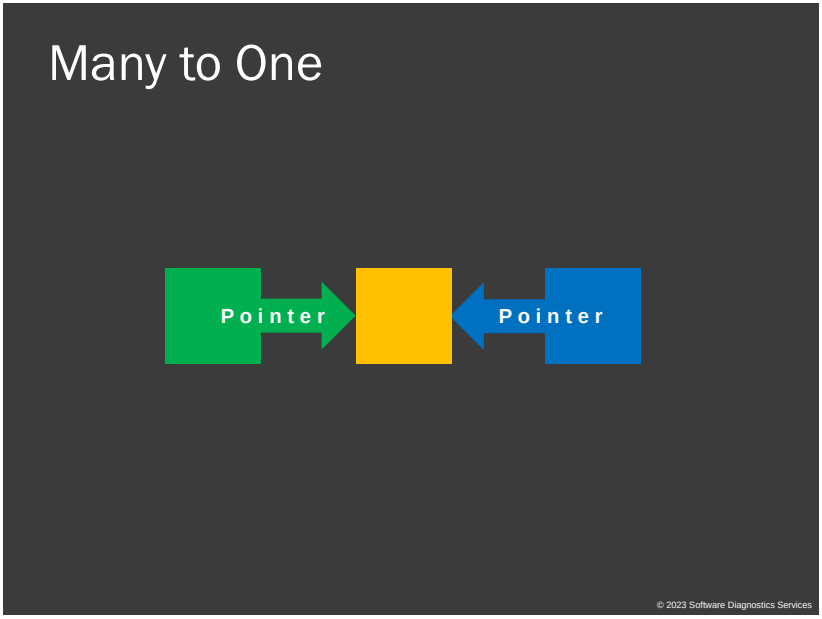
Conceptually, a pointer is an entity that refers (or points) to some other entity. We say entity, not an object, so as not to confuse it with objects in C++ or objects in object-oriented programming. This can be my finger, for example, pointing to an apple.

Pointer Dereference



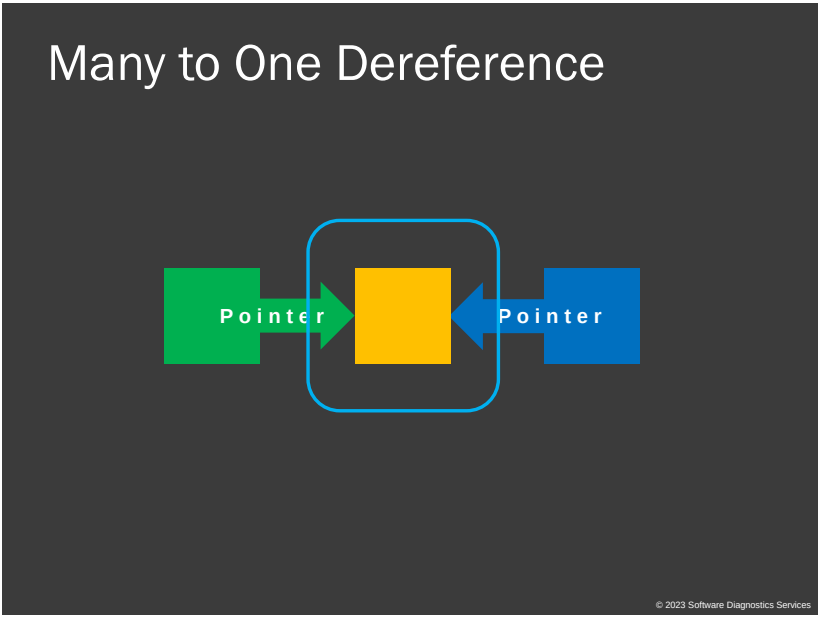
A pointer dereference is an act of getting the entity it references for further inspection or usage. Imagine I point to an apple, and you grab it to eat.

Many to One



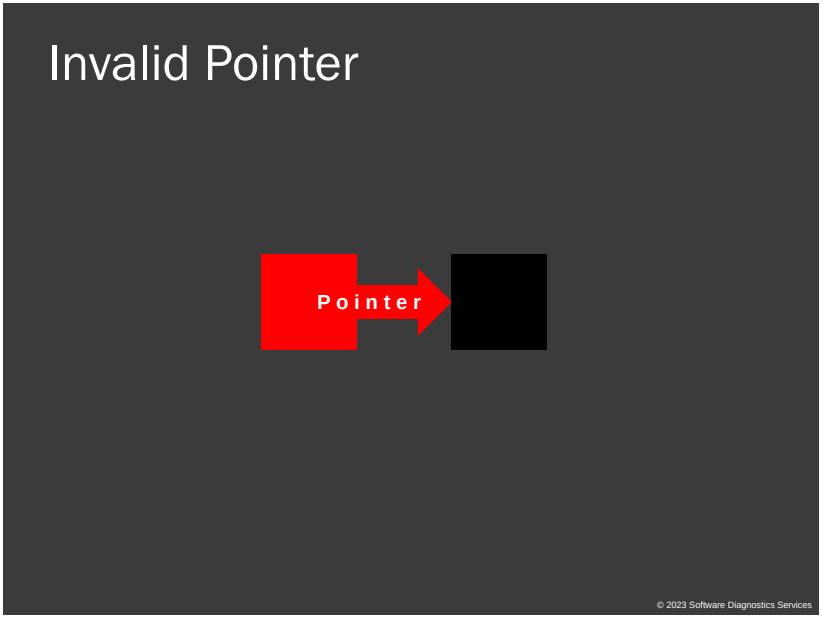
Several pointers can refer (or point) to the same entity. For example, two people are pointing to the same apple. So, conceptually, pointers are distinct from entities they point to. Should we call the latter pointees?

Many to One Dereference



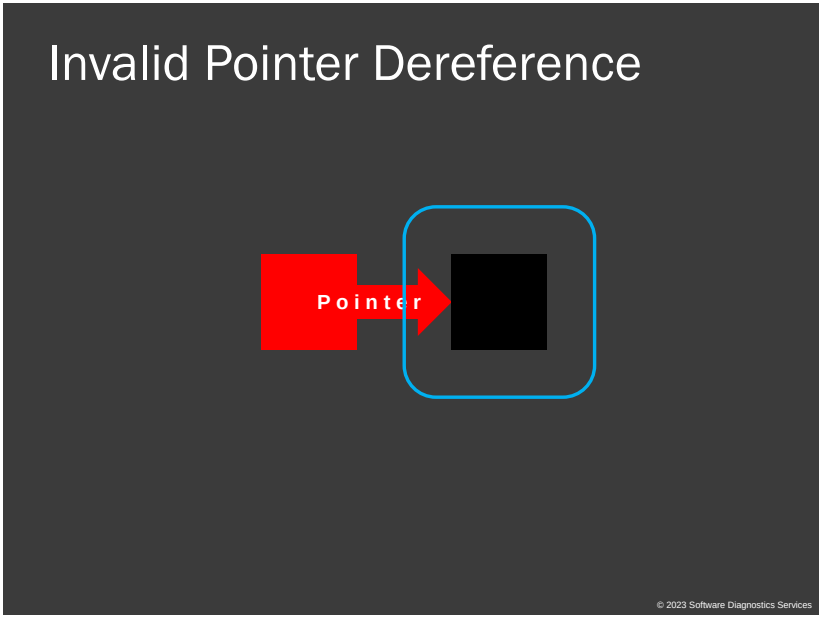
Of course, if you dereference the same object, you get the same object. If someone else grabs an apple, I point to, at the same time as you do, you both get the same apple.

Invalid Pointer



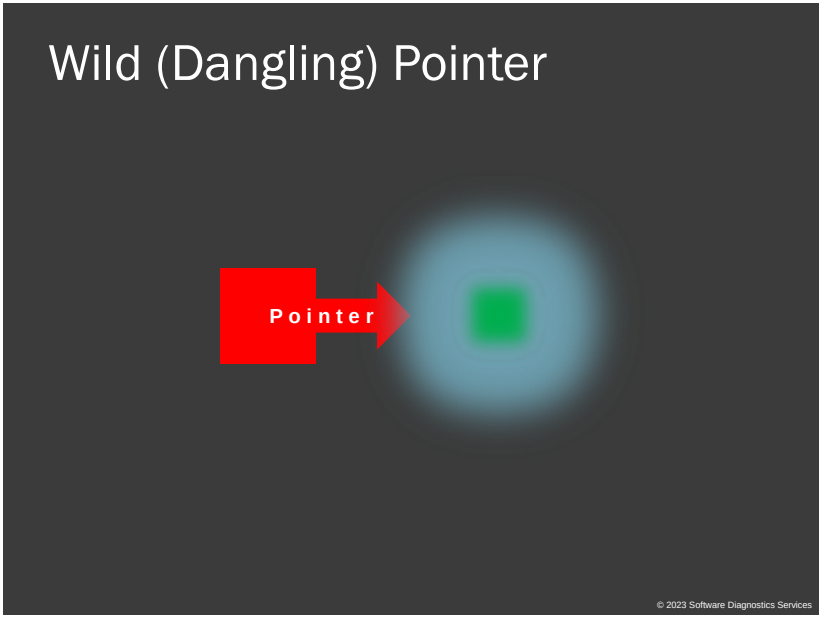
Some pointers may be invalid; for example, I may point to an imaginary apple.

Invalid Pointer Dereference



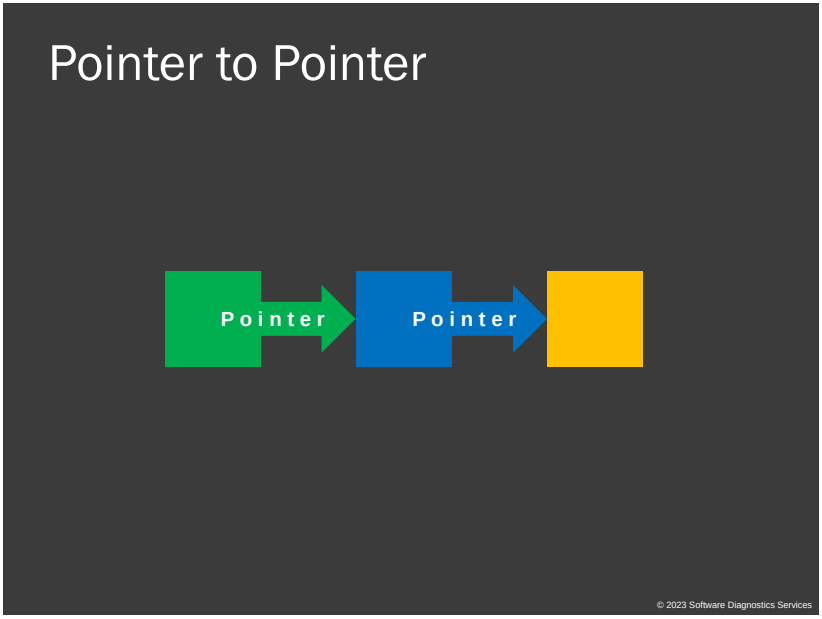
When you dereference an invalid pointer, you get a problem; for example, you fail to get an imaginary apple I point to.

Wild (Dangling) Pointer



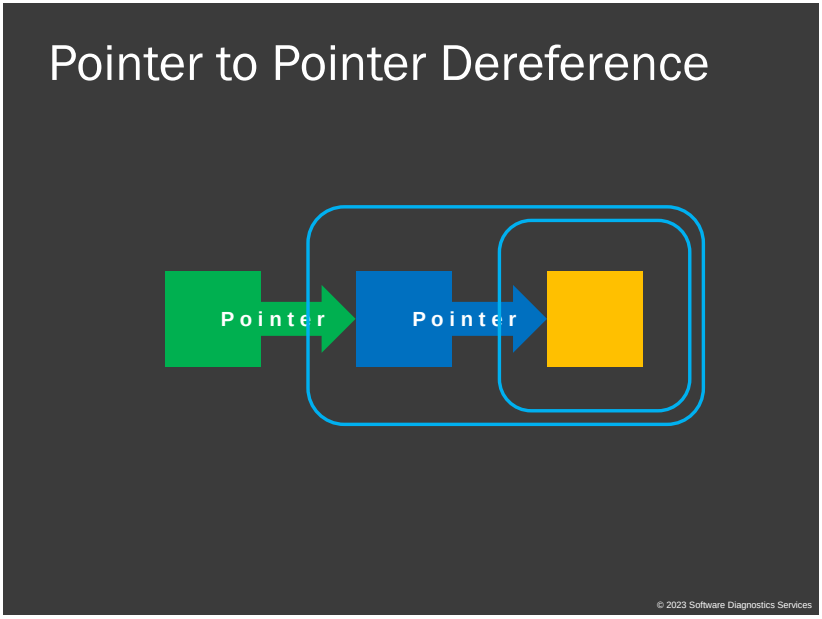
Some pointers are called dangling – they used to point to valid entities some time ago, but not anymore, so a dereference fails. You're reaching for an apple that I point to, but someone snatches it a split second ago.

Pointer to Pointer



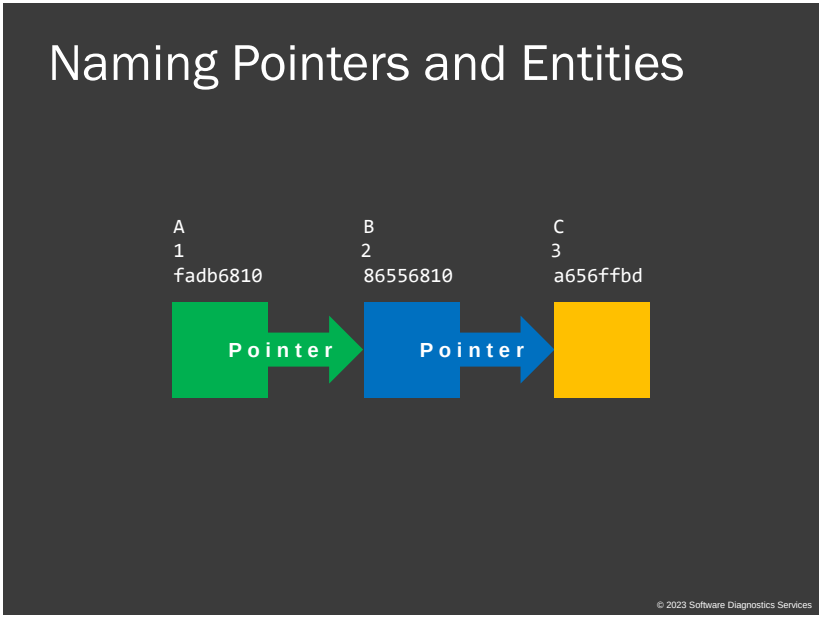
Since a pointer is also an entity that can be pointed to, there can be a chain of pointers. You point to me; I point to an apple.

Pointer to Pointer Dereference



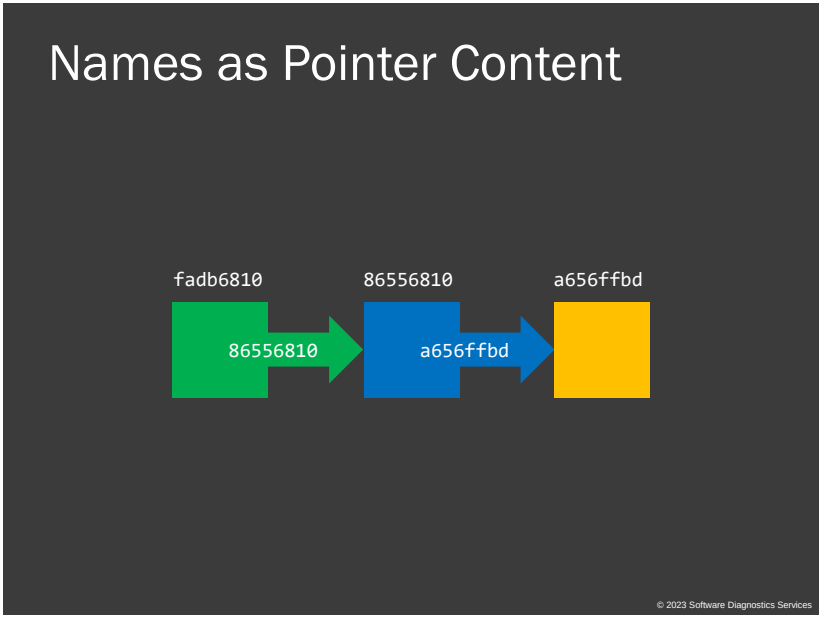
When we dereference the first pointer, we get an entity, another pointer, which we can also dereference to get the underlying entity. You point to me, but an alien snatches me with an apple I point to. Inside a ship, another alien takes an apple for analysis.

Naming Pointers and Entities



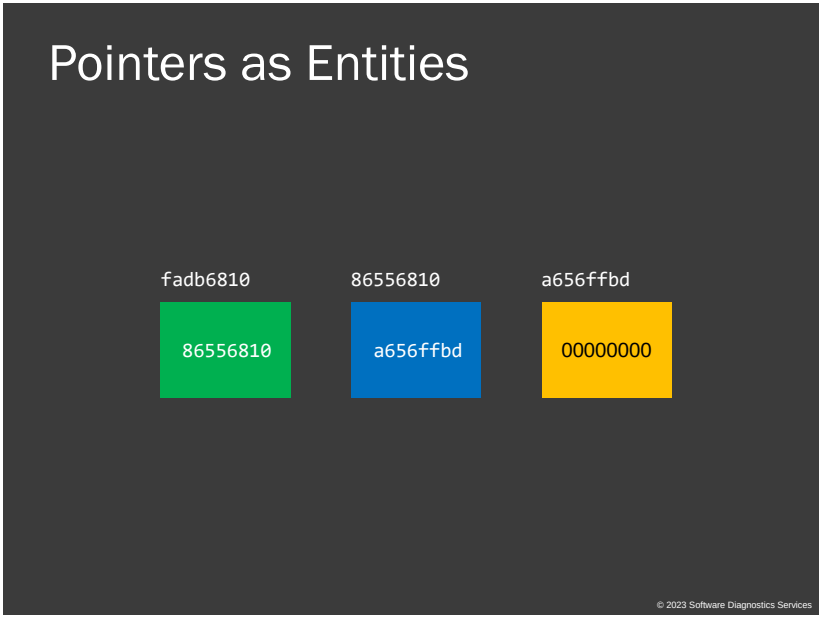
Names are distinct from entities. Names can be programming language identifiers or just unique numbers or IDs.

Names as Pointer Content



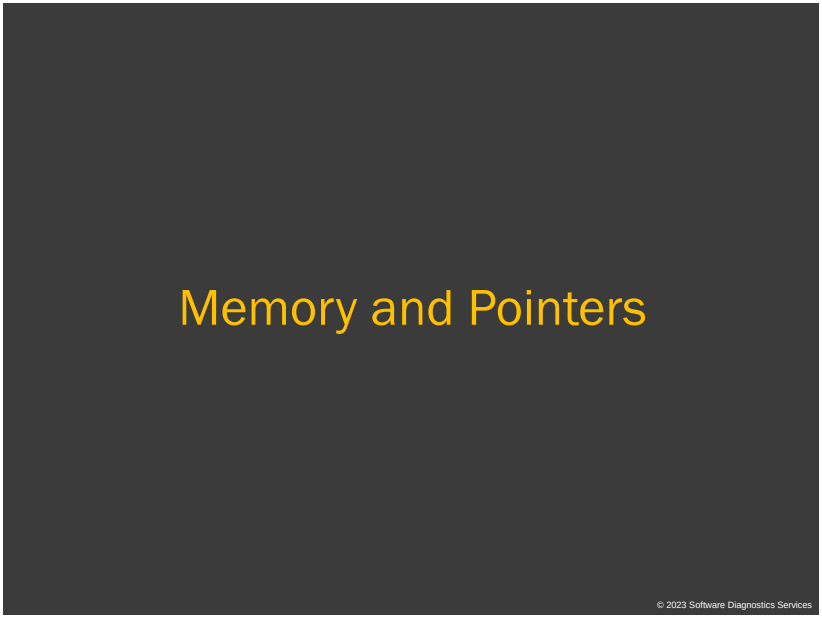
Pointers, as entities, may contain names, and these names may be names of pointers, too. If a pointer contains only a name, we say the pointer value is the name. So, the pointer value can be another pointer value, and the latter pointer value is the name of some other entity.

Pointers as Entities



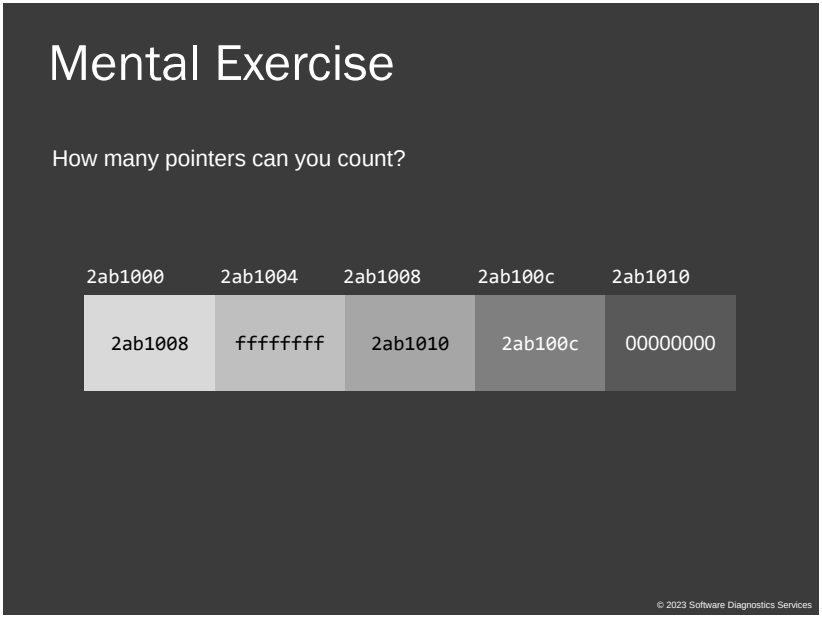
Pointer dereference is an act. If we put acts aside, pointers are just entities with some content that can be interpreted as a name if necessary. All these dereferences happen only at runtime. The pointer content (its value) may be invalid for all time without any problem until we use it.

Memory and Pointers



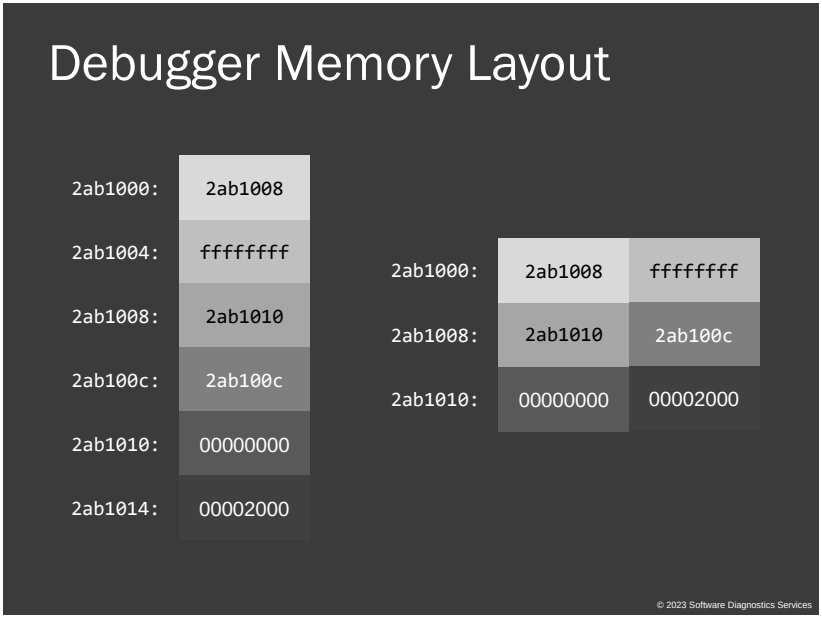
Now, we look at the memory representation of pointers and entities they point to.

Mental Exercise



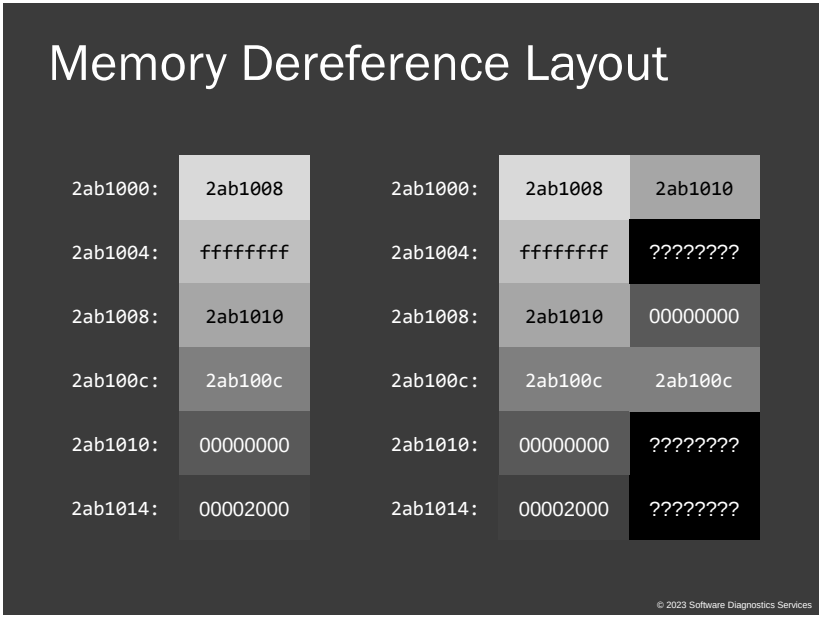
Here, in this picture, entities are the so-called memory cells. Memory cells have addresses that start from 0 and are usually incremented by the so-called pointer size, which is 4 on 32-bit systems and 8 on 64-bit systems. Here, for visual clarity, we use memory cells from a 32-bit system.

Debugger Memory Layout



When we use a debugger, it prints memory cell addresses and their contents in a certain layout shown on this slide. Some debugger commands, such as **x** in GDB, use 2-column and some n-column layouts to print memory.

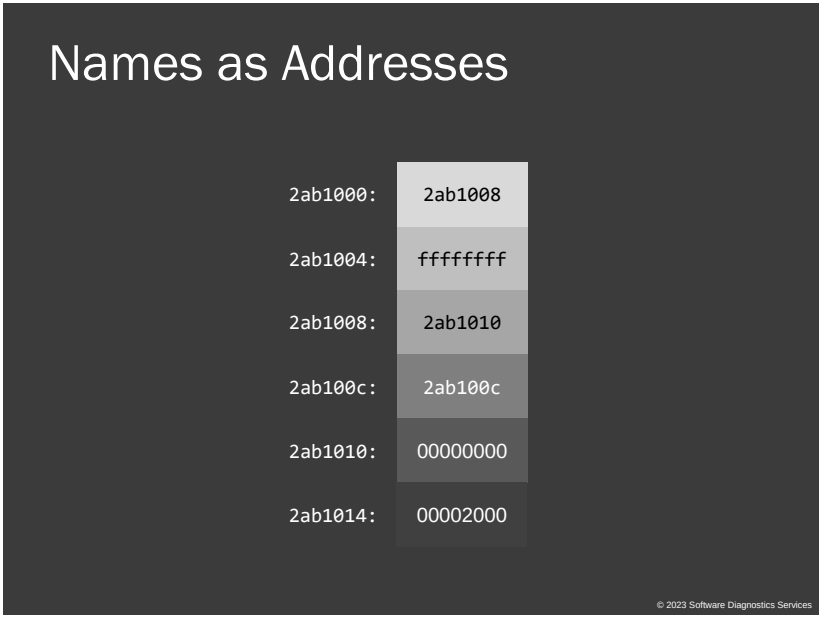
Memory Dereference Layout



For a 2-column format, some debuggers and their commands may interpret the second column as a pointer. In such a case, the third column is a value from a pointer dereference. Also, notice a case when a pointer points to itself. For GDB, it is possible to emulate such behavior using a custom script:

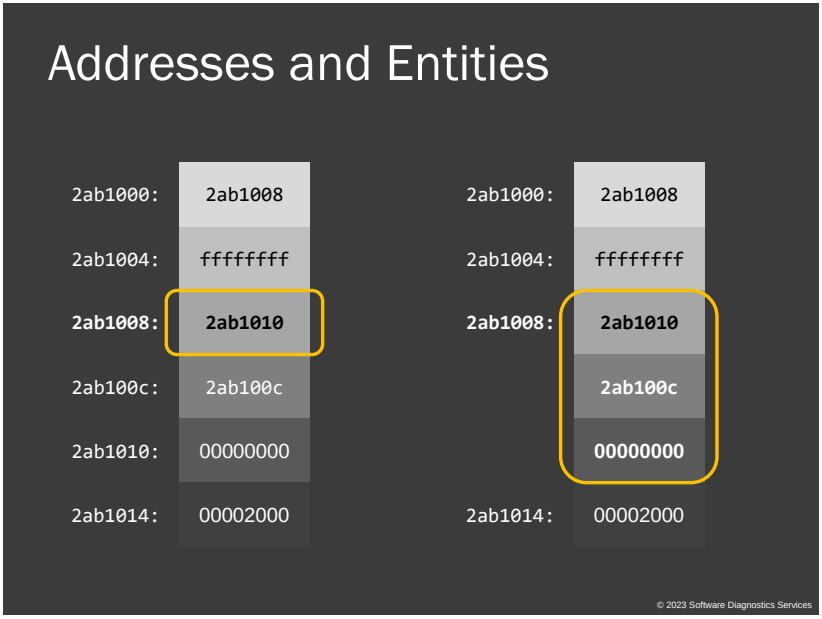
```
define dpp
    set $i = 0
    set $p = $arg0
    while $i < $arg1
        printf "%p: ", $p
        x/gx *(long *)$p
        set $i = $i + 1
        set $p = $p + 8
    end
end
```

Names as Addresses



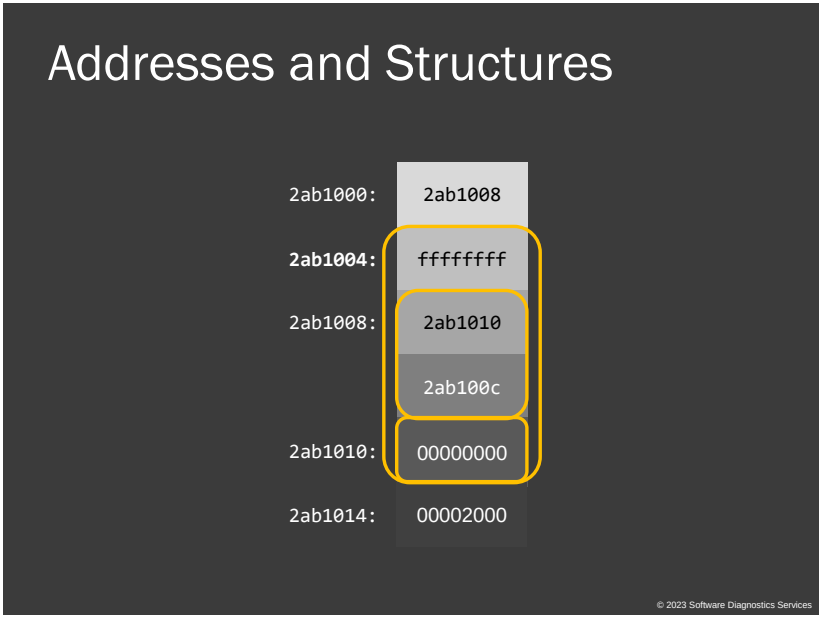
To repeat, for memory layout, names are interpreted as addresses, and memory cell content (cell value) can also be interpreted as a memory address.

Addresses and Entities



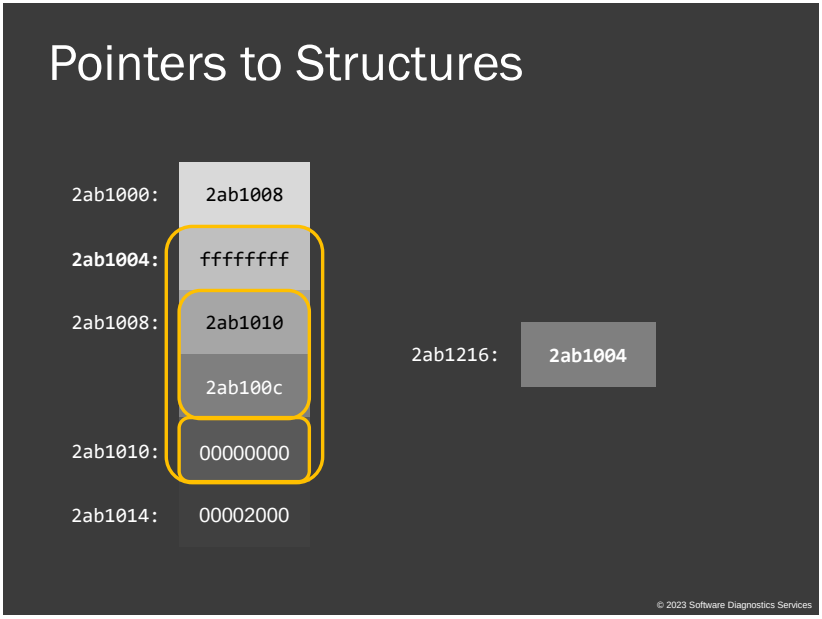
Entities can be either single cells or multicells. Each part of a multicell can be interpreted as a memory address, if necessary, even if it wasn't meant to be a memory address.

Addresses and Structures



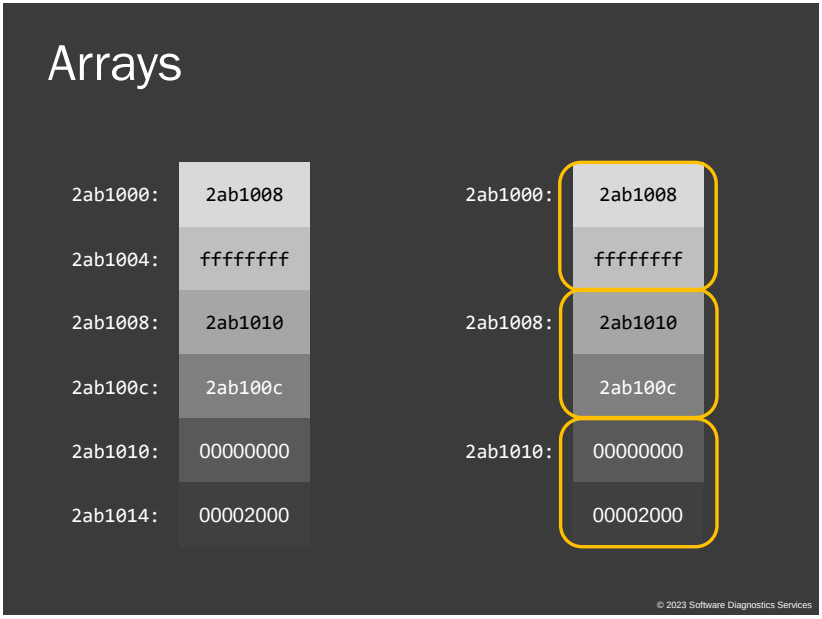
A structure in memory is a sequential collection of memory cells; some may be multicell and themselves substructures. Each part of a structure, its member, or structure field has its own address as well, in addition to the overall address of the structure.

Pointers to Structures



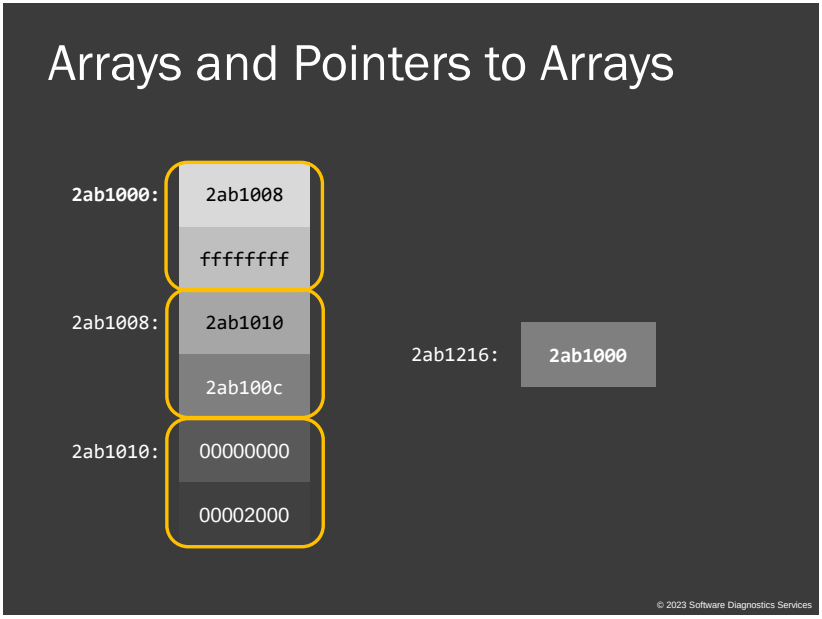
A structure has its address. A pointer to a structure is a memory cell that contains that address. It has its own address.

Arrays



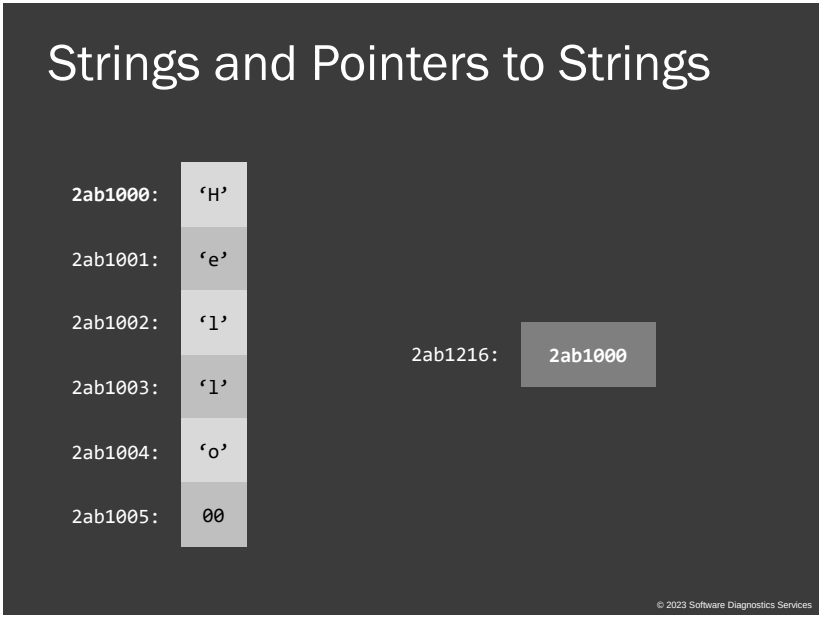
An array is a contiguous sequence of n-cells in memory called array elements. Each array element has its own address. Since the size of each array element is fixed and the same, addressing the random element is fast.

Arrays and Pointers to Arrays



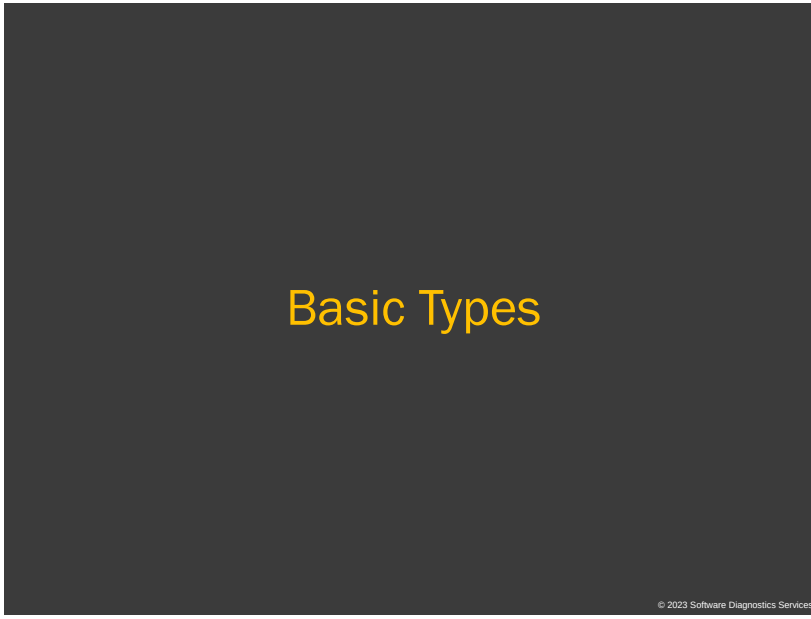
The array address is the address of its first element. But a pointer to an array is a different memory cell that contains the array address. This is similar to structures and pointers to structures. An array can be considered as a structure as well.

Strings and Pointers to Strings



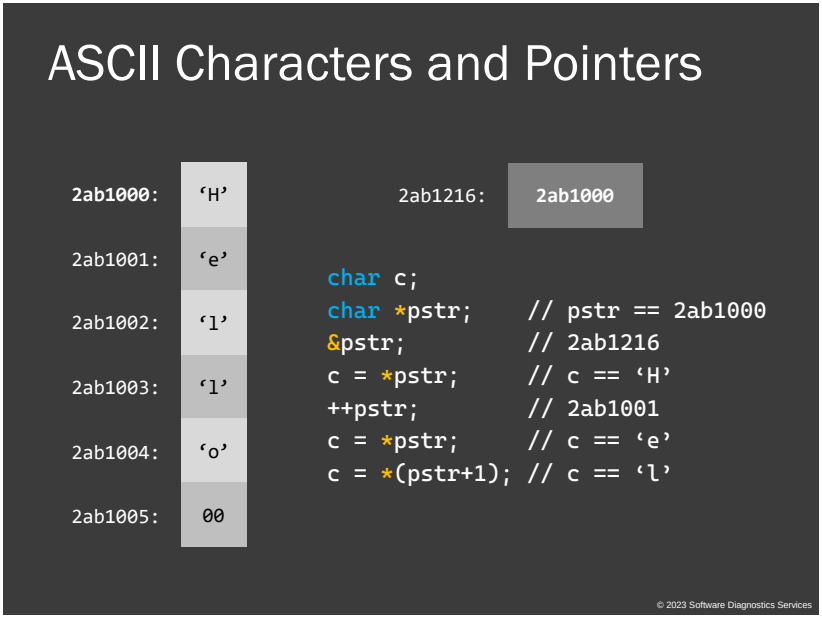
What about strings? An ASCII string is a zero-terminated array of one-byte memory cells. The address of a string is the address of its first byte. Similar to arrays, a pointer to a string is a memory cell that contains the address of the string, the address of its first element – its first character.

Basic Types



Now, we look at a few fundamental basic types.

ASCII Characters and Pointers

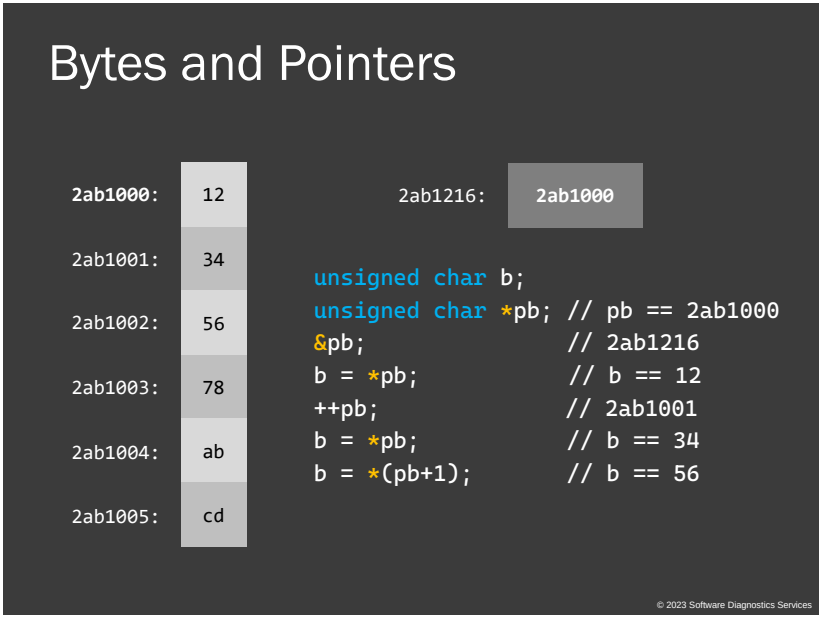


We have already looked at ASCII zero-terminated strings and pointers conceptually using memory diagrams. Here, we look at some idiomatic C code.

The code example corresponding to the memory diagram:

```
char c;
char *pstr;     // pstr == 2ab1000
&pstr;          // 2ab1216
c = *pstr;      // c == 'H'
++pstr;         // 2ab1001
c = *pstr;      // c == 'e'
c = *(pstr+1); // c == 'l'
```

Bytes and Pointers

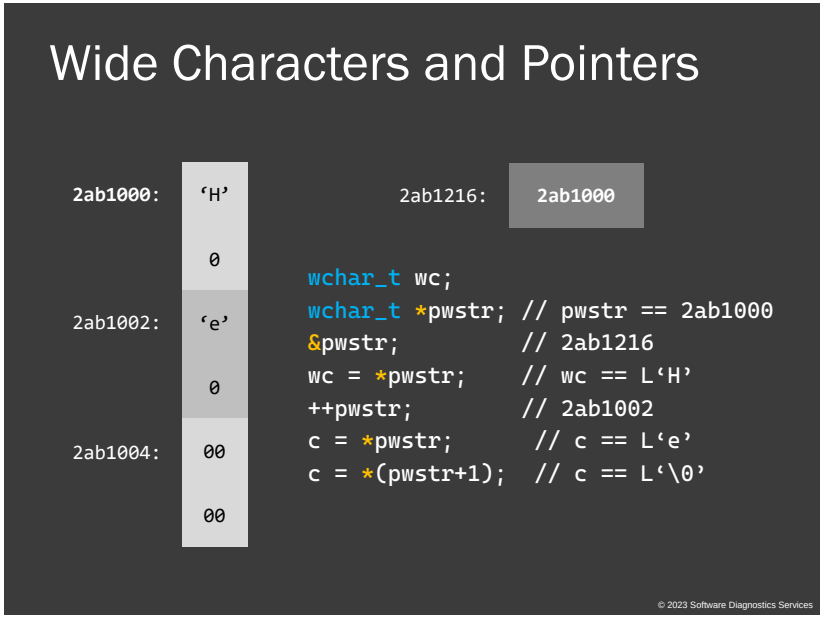


Characters are signed with small integer values from -128 to 127. But if we want to work with bytes with unsigned values from 0 to 255, we need to use unsigned characters. Later, we see what other types are available to work with bytes.

The code example corresponding to the memory diagram:

```
unsigned char b;
unsigned char *pb; // pb == 2ab1000
&pb;               // 2ab1216
b = *pb;           // b == 12
++pb;              // 2ab1001
b = *pb;           // b == 34
b = *(pb+1);       // b == 56
```

Wide Characters and Pointers

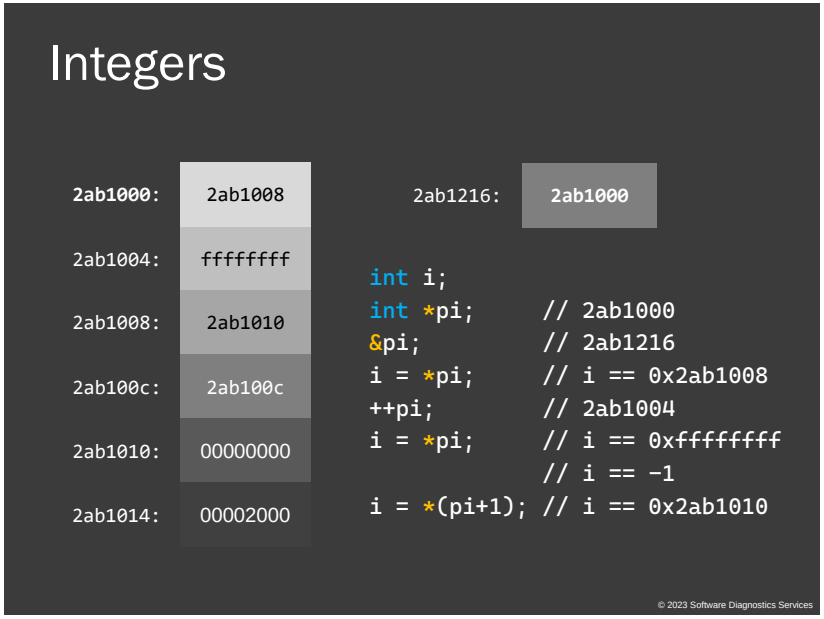


If you want to use UNICODE, it is natural to use wide characters that occupy two bytes each.

The code example corresponding to the memory diagram:

```
wchar_t wc;
wchar_t *pwstr; // pwstr == 2ab1000
&pwstr;         // 2ab1216
wc = *pwstr;    // wc == L'H'
++pwstr;        // 2ab1002
c = *pwstr;     // c == L'e'
c = *(pwstr+1); // c == L'\0'
```

Integers

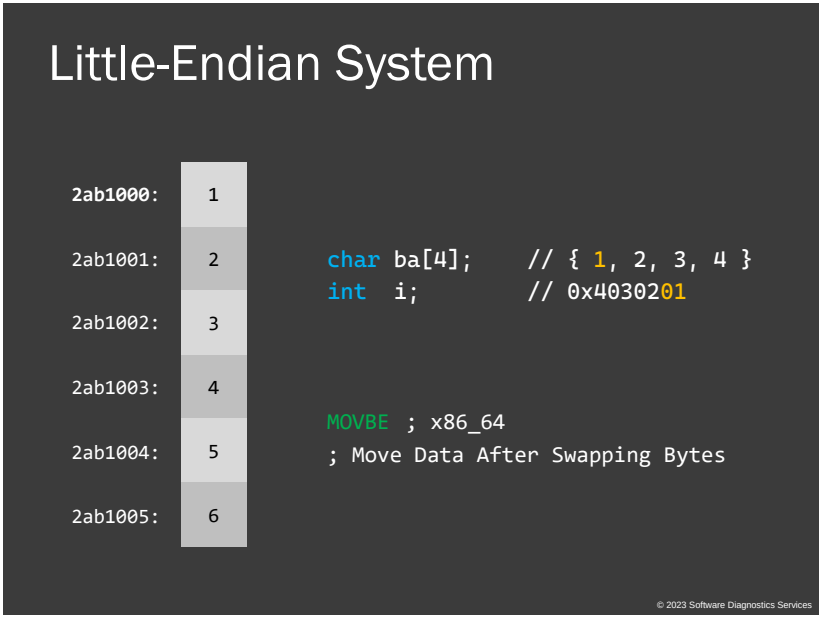


The second type we look at now is integers, which occupy 4 bytes.

The code example corresponding to the memory diagram:

```
int i;
int *pi;      // 2ab1000
&pi;          // 2ab1216
i = *pi;      // i == 0x2ab1008
++pi;         // 2ab1004
i = *pi;      // i == 0xffffffff
              // i == -1
i = *(pi+1); // i == 0x2ab1010
```

Little-Endian System

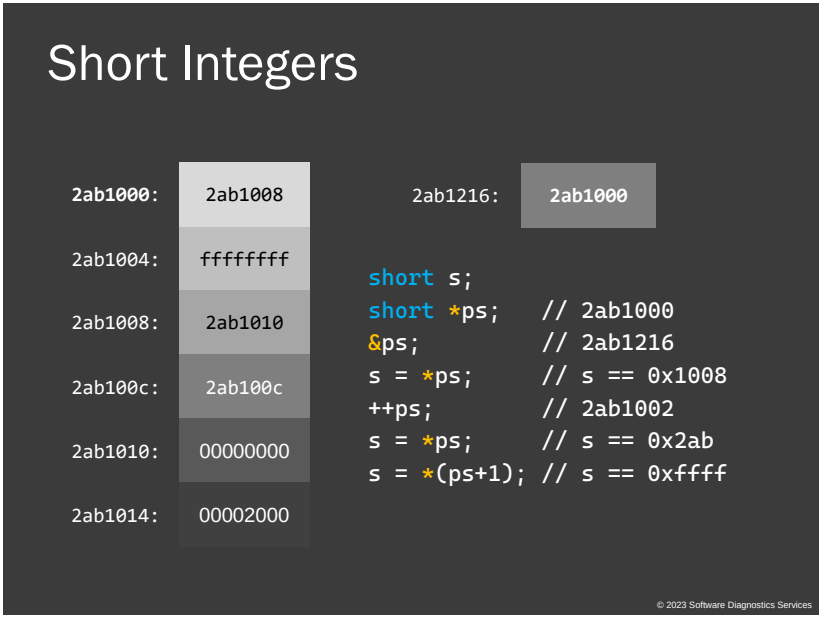


When converting between byte sequences and number values, we need to consider the little-endian system where the least significant digits reside at the lowest memory addresses.

The code example corresponding to the memory diagram:

```
char ba[4];     // { 1, 2, 3, 4 }
int  i;         // 0x4030201
```

Short Integers

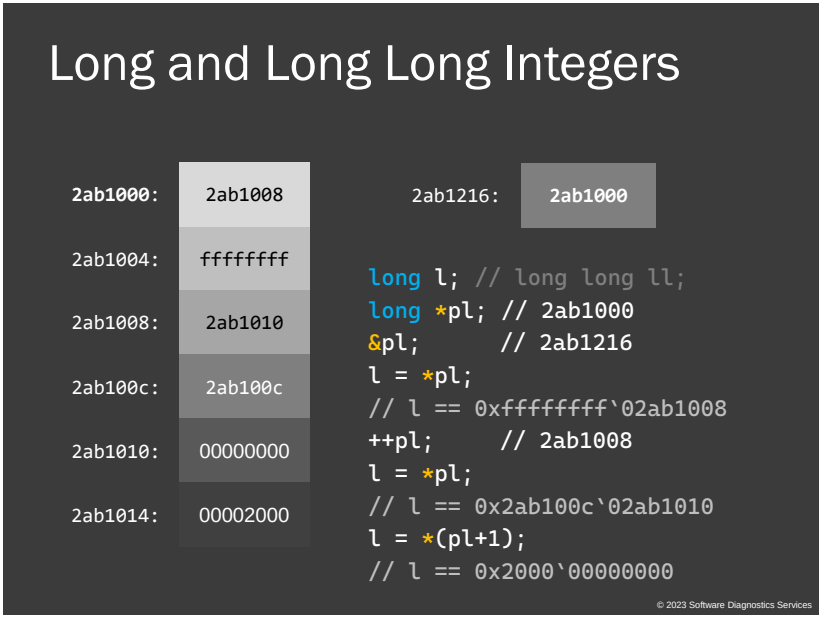


Short integers occupy 2 bytes.

The code example corresponding to the memory diagram:

```
short s;
short *ps;   // 2ab1000
&ps;         // 2ab1216
s = *ps;     // s == 0x1008
++ps;        // 2ab1002
s = *ps;     // s == 0x2ab
s = *(ps+1); // s == 0xffff
```

Long and Long Long Integers

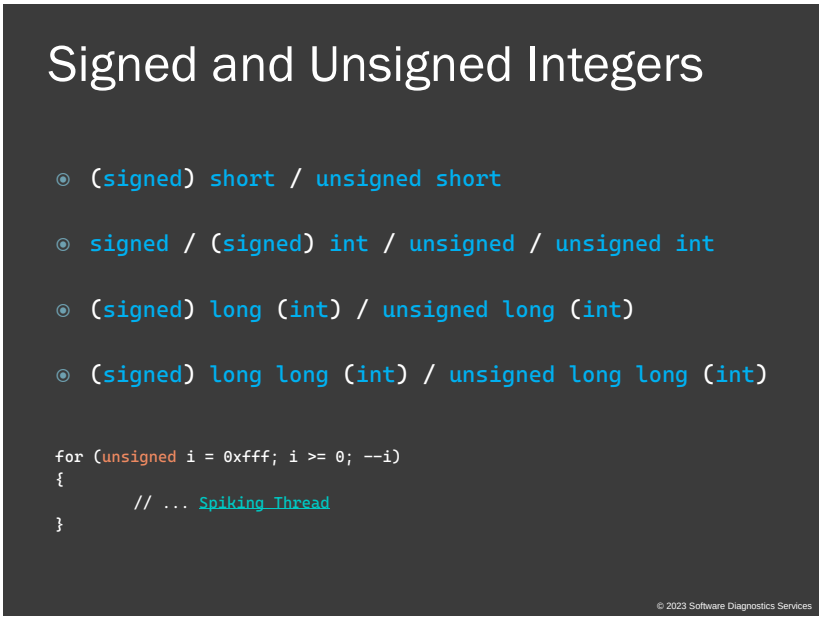


If we want 8-byte 64-bit integers, we need to use `long` or `long long` for portability.

The code example corresponding to the memory diagram:

```
long l;         // long long ll;
long *pl;       // 2ab1000
&pl;            // 2ab1216
l = *pl;        // ll == 0xffffffff`02ab1008
++pl;           // 2ab1008
l = *pl;        // ll == 0x2ab100c`02ab1010
l = *(pl+1);    // ll == 0x2000`00000000
```

Signed and Unsigned Integers

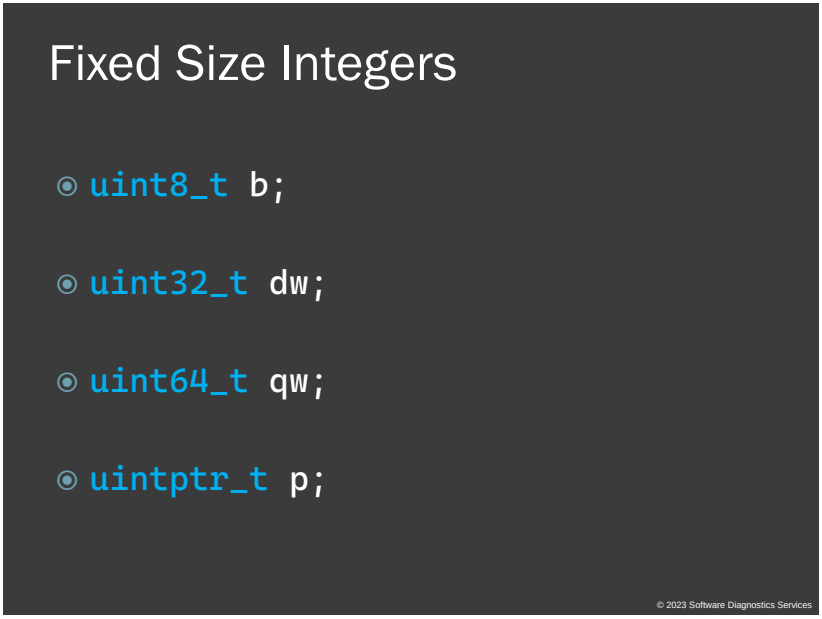


We need to be careful to use unsigned index variables in classic loops. The following code example loops indefinitely since the loop variable is always positive:

```
for (unsigned i = 0xfff; i >= 0; --i)
{
      // ... Spiking Thread
}
```

Spiking Thread memory analysis pattern
https://www.dumpanalysis.org/blog/index.php/2015/12/14/crash-dump-analysis-patterns-part-14-linux/

Fixed Size Integers

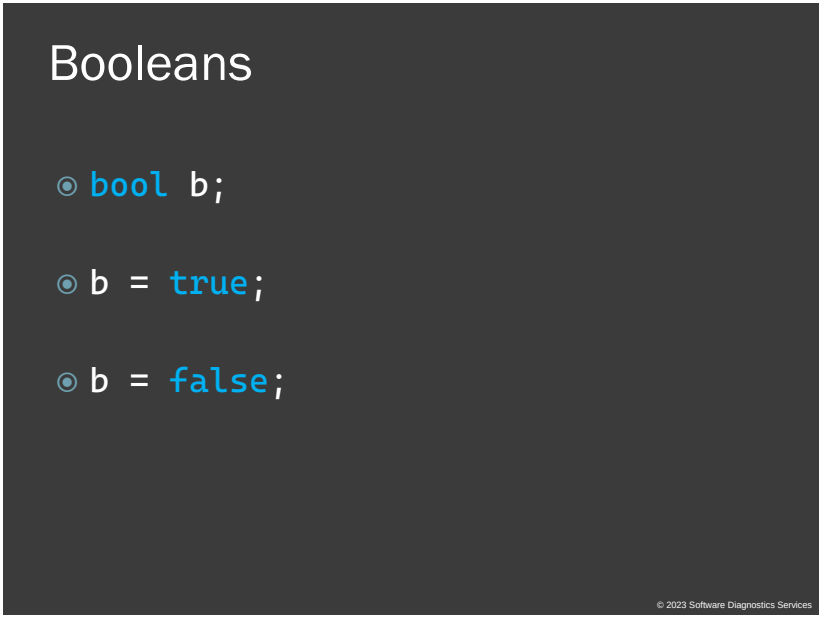


It is also possible to be precise and use portable fixed-size types.

The code example:

```
uint8_t   b;
uint32_t  dw;
uint64_t  qw;
uintptr_t p;
```

Booleans

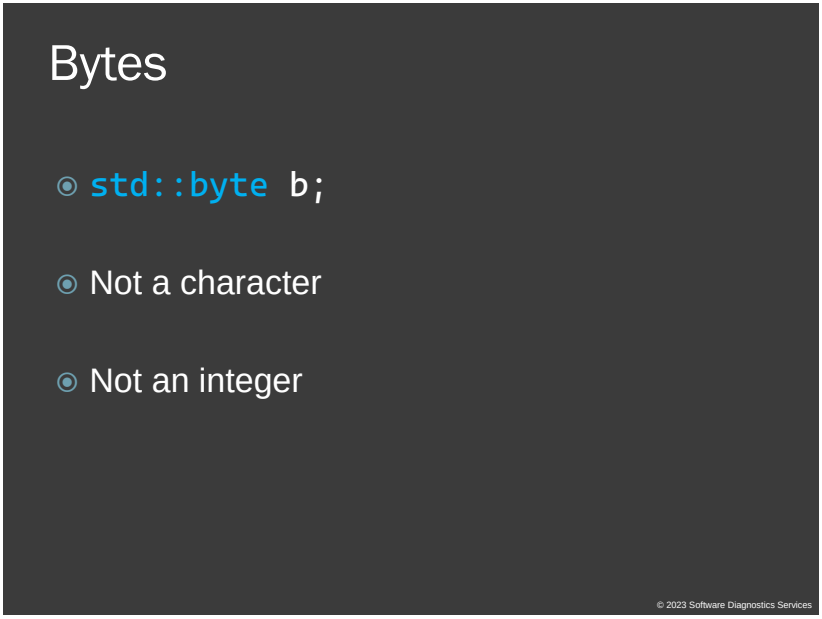


C++ also includes a native type for boolean variables. If you want to use it in pure C, you need to include the *stdbool.h* header.

The code example:

```
bool b;

b = true;
b = false;
```

Bytes

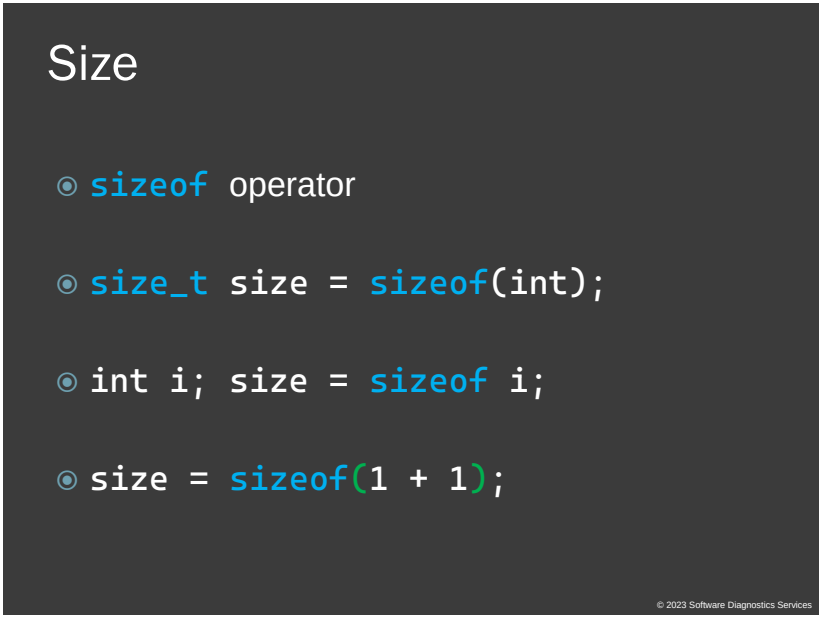


The latest C++ standards also include a distinct type for bytes.

The code example:

```
std::byte b;
```

Size

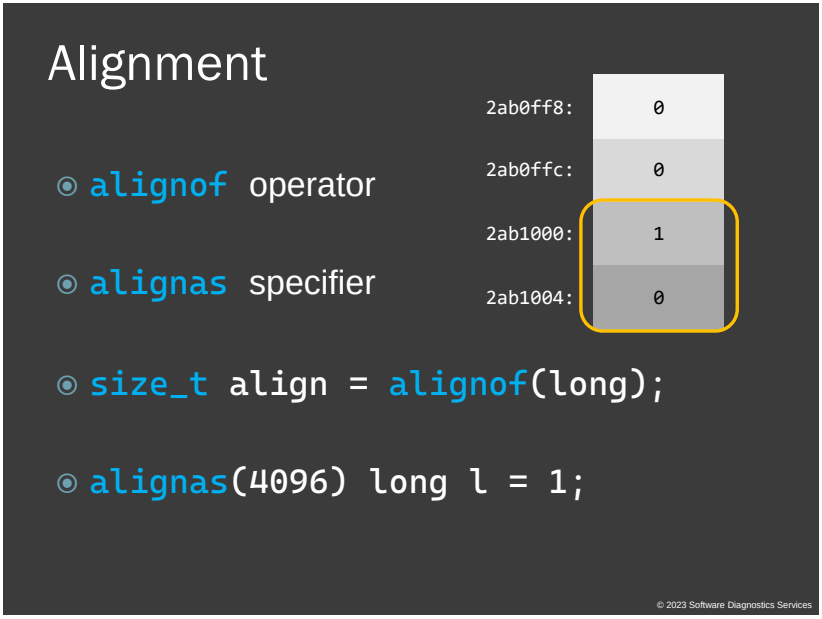


The `sizeof` operator can evaluate the size of types, variables, and target result types of expressions (without expression evaluation).

The code example:

```
size_t size = sizeof(int);

int i;

size = sizeof i;
size = sizeof(1 + 1);
```

Alignment

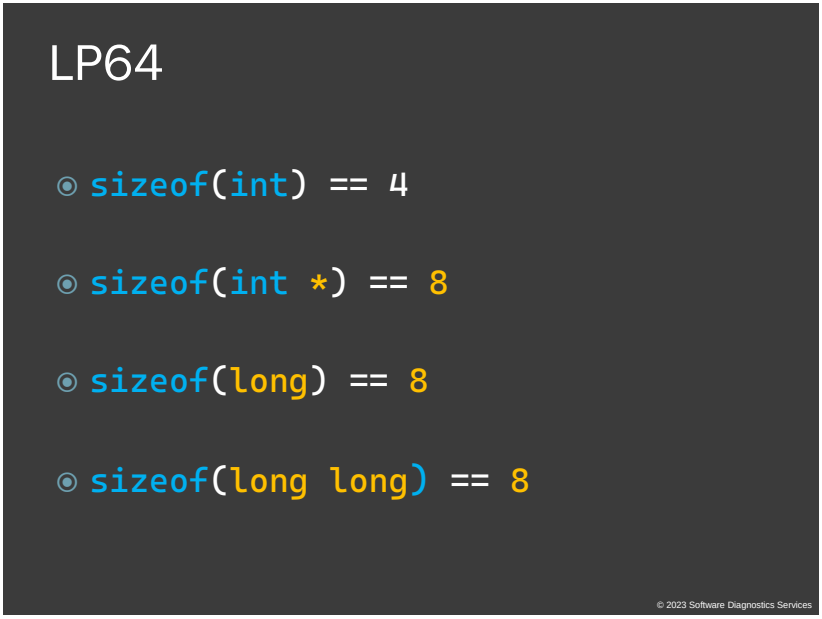


Variables are usually aligned in memory at offsets divisible by their type size value in bytes. You can get default alignment values using the `alignof` operator and change the default alignment using the `alignas` specifier.

The code example:

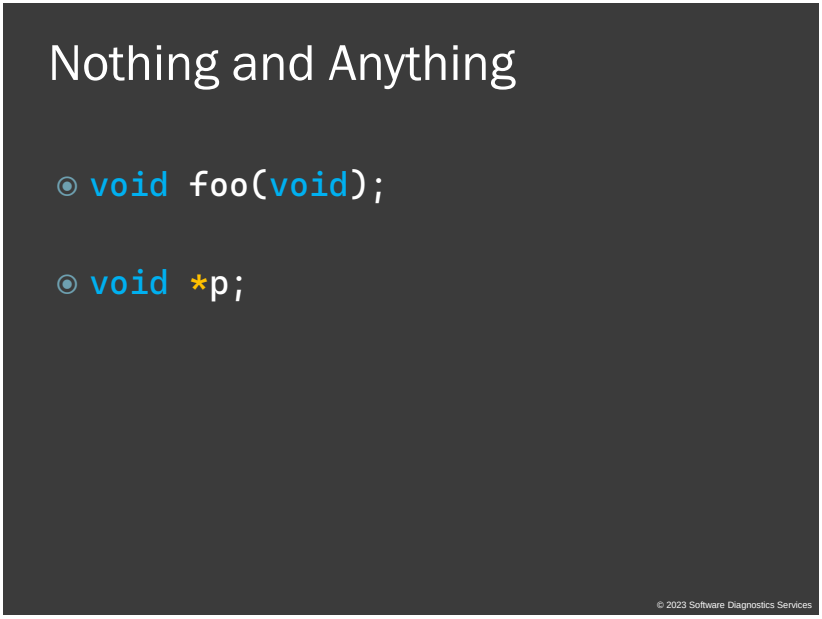
```
size_t align = alignof(long);

alignas(4096) long l = 1;
```

LP64



Linux uses the so-called LP64 data model, where `long` integers and pointers are 64-bit.

Nothing and Anything

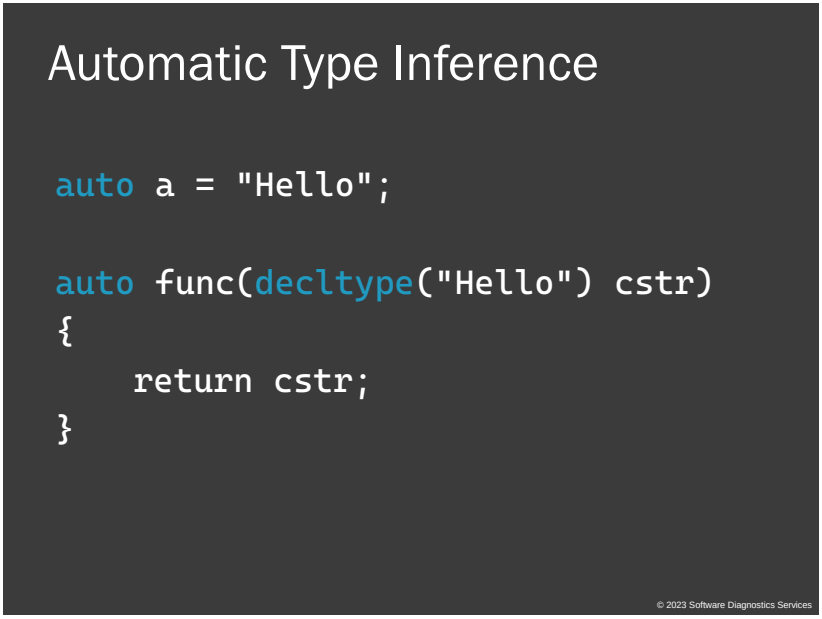


Two distinct types correspond to the concepts of *Nothing* and *Anything* you can find in other programming languages: `void` and `void *`. The latter is a pointer to any type.

The code example:

```
void foo(void);
void *p;
```

Automatic Type Inference



C++11 added automatic type specification, so the type is deduced from the initializing expression.

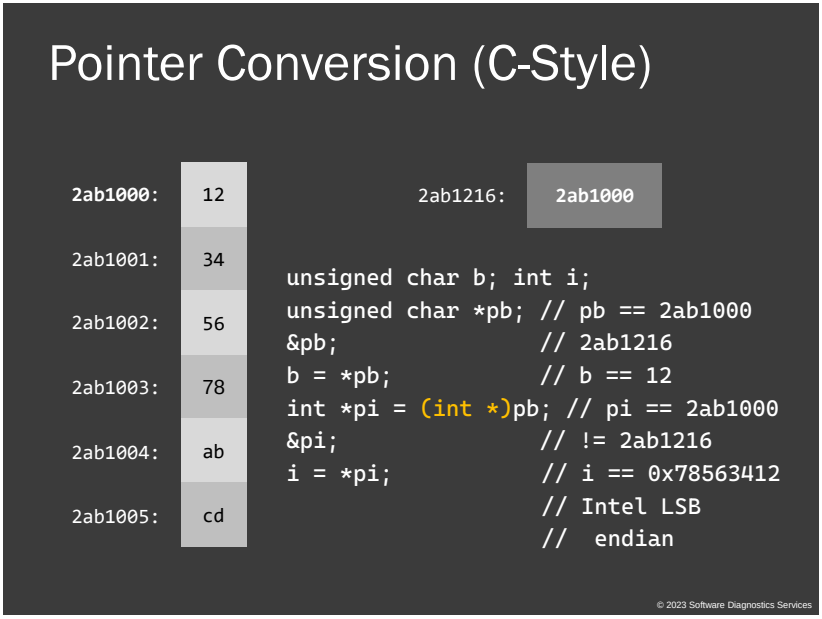
The code example:

```
auto a = "Hello";

auto func(decltype("Hello") cstr)
{
      return cstr;
}
```

Entity Conversion

As you anticipate, the same memory cell addresses and their values are the basis of conversion between different entity types. So, let's look at some examples.

Pointer Conversion (C-Style)

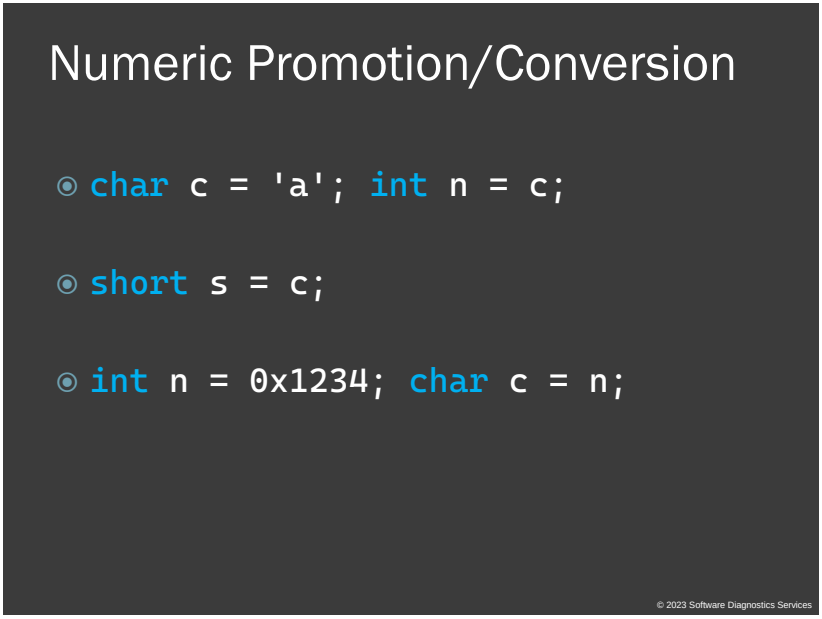


Pointers can be converted to each other freely because their value is just a memory address. However, when we dereference them, we get the value based on underlying memory contents, which don't change as illustrated here. Please also note that due to the least significant byte endian convention, the integer value we get differs from the memory layout byte order.

The code example corresponding to the memory diagram:

```
unsigned char b; int i;
unsigned char *pb;      // pb == 2ab1000
&pb;                    // 2ab1216
b = *pb;                // b == 12
int *pi = (int *)pb;    // pi == 2ab1000
&pi;                    // != 2ab1216
i = *pi;                // i == 0x78563412
                        // Intel LSB endian
```

Numeric Promotion/Conversion

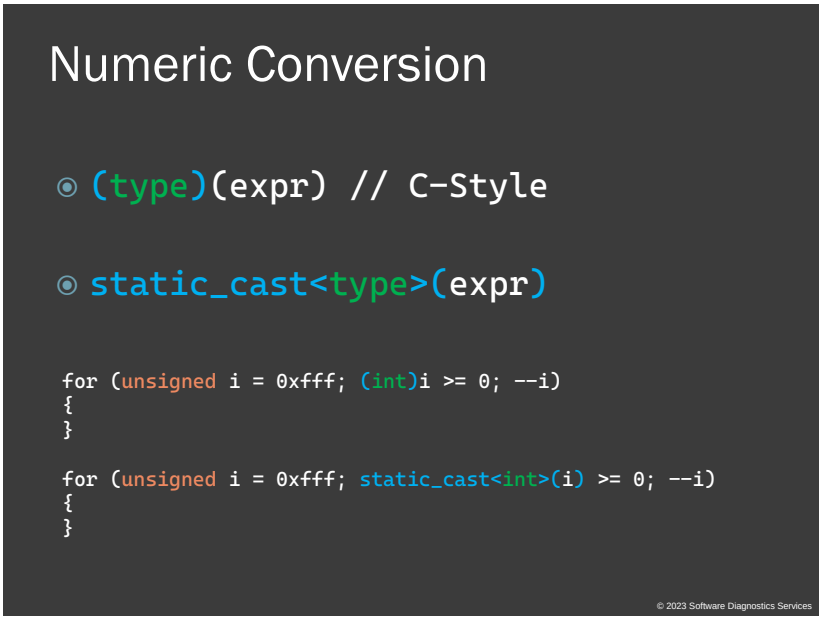


Values from the lesser range of values can be automatically promoted to types with a wider range of values. The opposite automatic conversion may lose some bits of information and should be carefully reviewed.

The code example:

```
char c = 'a';
int n = c;

short s = c;

int n = 0x1234;
char c = n;
```

Numeric Conversion

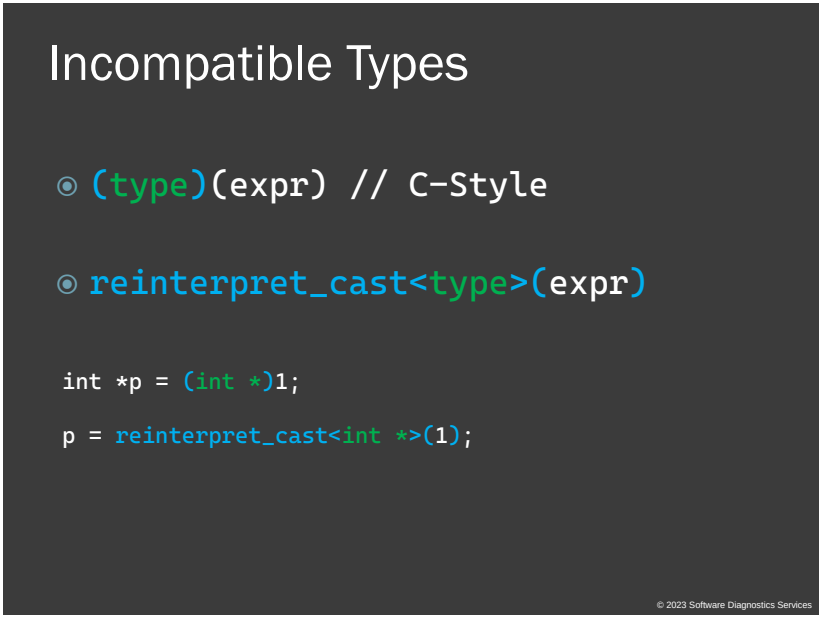


In the absence of automatic conversion for compatible types, we can use C-style casts or explicit, specific C++ casts.

The following code examples solve the problem with the infinite loop:

```
for (unsigned i = 0xfff; (int)i >= 0; --i)
{
}

for (unsigned i = 0xfff; static_cast<int>(i) >= 0; --i)
{
}
```

Incompatible Types

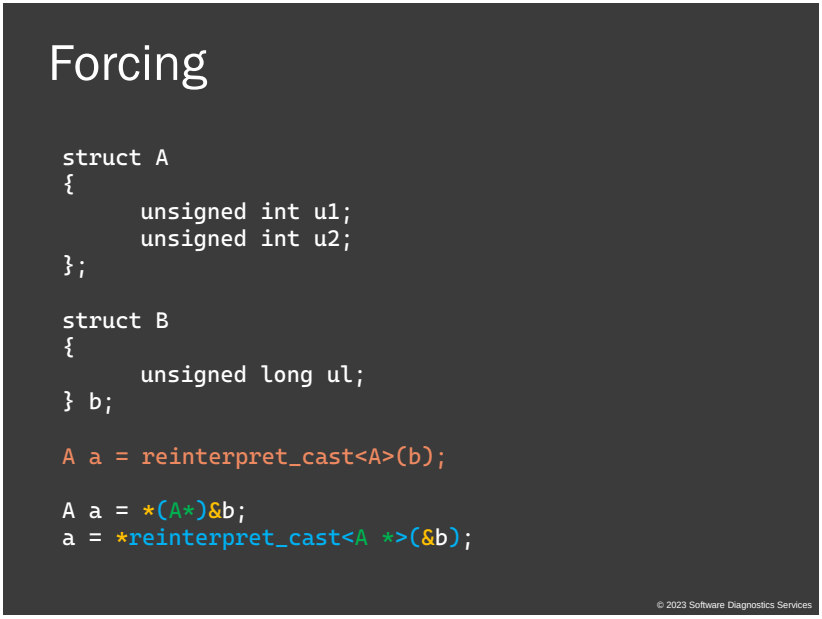


When types are incompatible, for example, integers and pointers to them, we can use either C-style casts or the specific C++ type reinterpretation cast.

The code example:

```
int *p = (int *)1;

p = reinterpret_cast<int *>(1);
```

Forcing

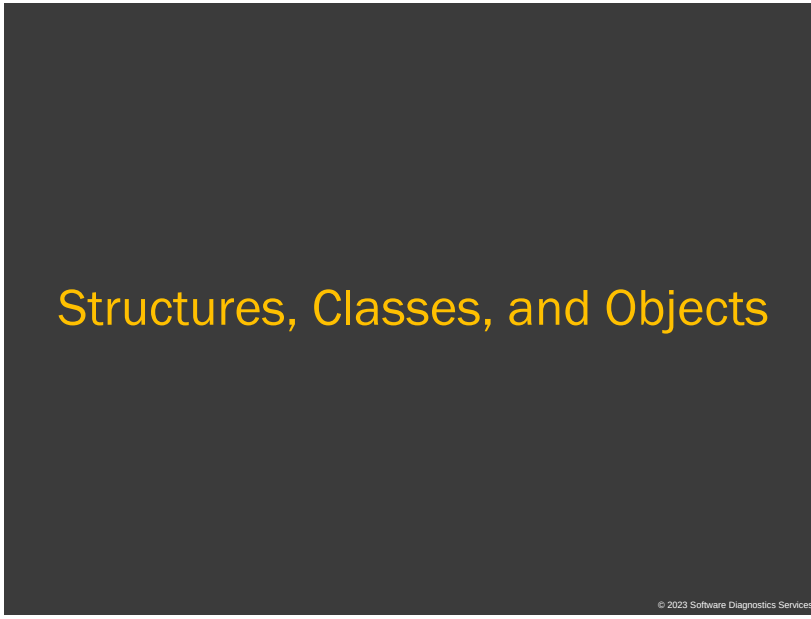


Different structures are even more incompatible with the failing direct C++ reinterpretation cast. However, we can force reinterpretation of structures by reinterpreting a pointer to a source structure as a pointer to a target structure and then dereferencing it. In such a case, the underlying memory cells are reinterpreted as the target structure field values. You can review the code example after studying the next two sections on structures and memory:

```
struct A
{
      unsigned int u1;
      unsigned int u2;
};

struct B
{
      unsigned long ul;
} b;
```

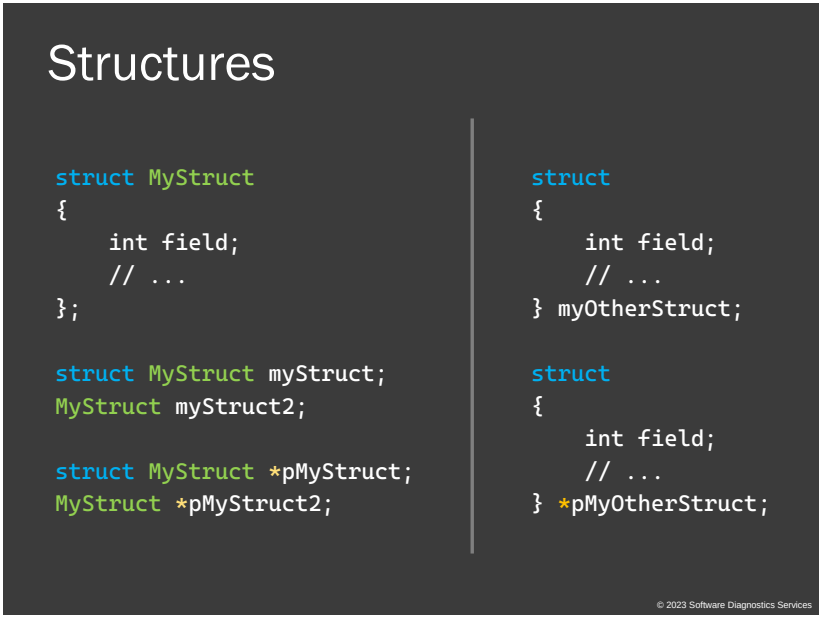
```
A a = reinterpret_cast<A>(b);

A a = *(A*)&b;

a = *reinterpret_cast<A *>(&b);
```

Structures, Classes, and Objects



Now, we cover structures, classes, and their objects.

Structures



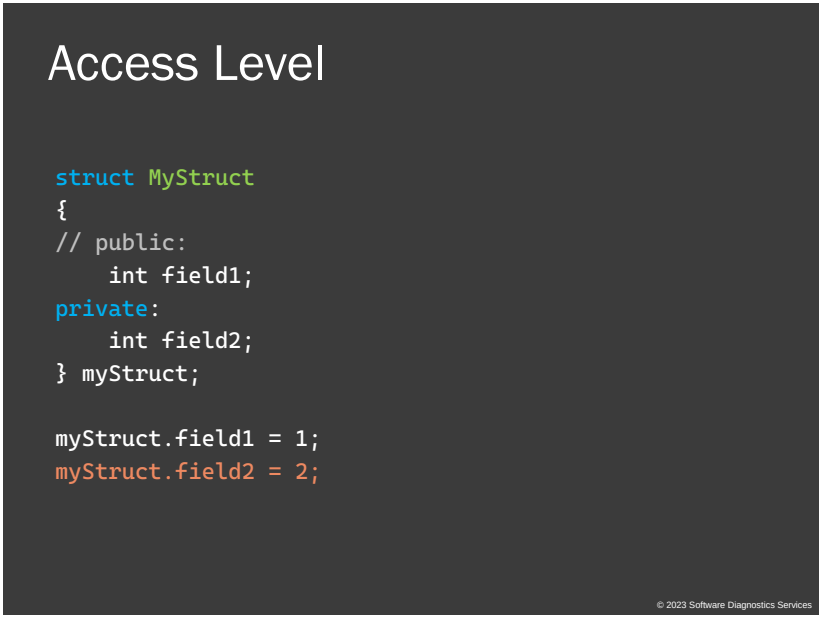
We can view structures as collections of fields laid out in memory. Structures may have names or can be anonymous, as on the right.

The code example:

```
struct MyStruct
{
    int field;
    // ...
};

struct MyStruct myStruct;
MyStruct myStruct2;

struct MyStruct *pMyStruct;
MyStruct *pMyStruct2;
```

```
struct
{
    int field;
    // ...
} myOtherStruct;

struct
{
    int field;
    // ...
} *pMyOtherStruct;
```

Access Level

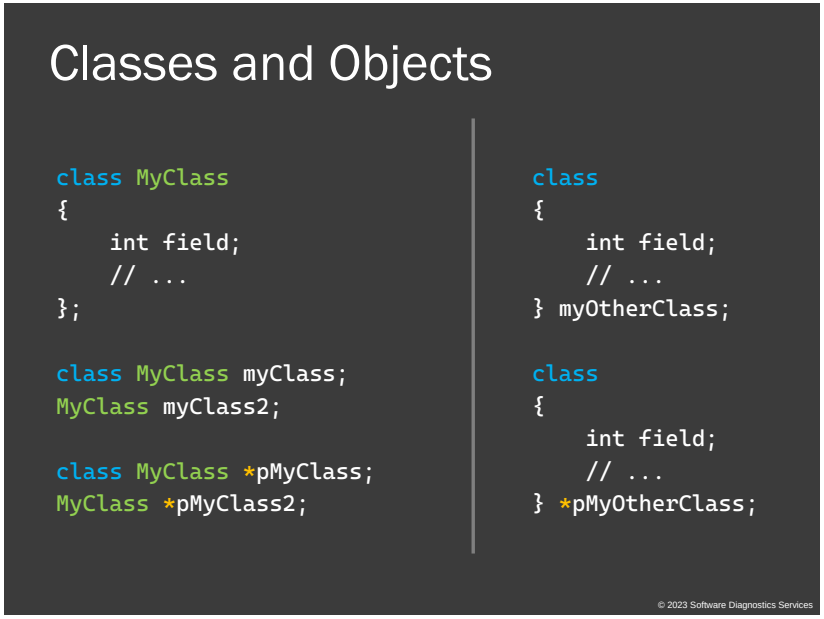


Fields with the `private` access specifier cannot be referenced from the outside.

The code example:

```
struct MyStruct
{
// public:
    int field1;
private:
    int field2;
} myStruct;

myStruct.field1 = 1;
myStruct.field2 = 2;
```

Classes and Objects

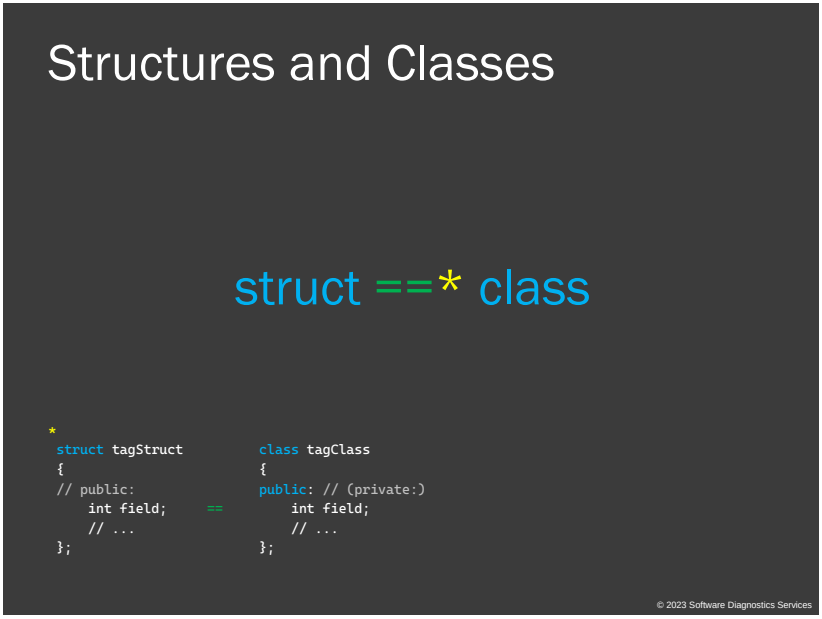


Classes have the same structure.

The code example:

```
class MyClass
{
    int field;
    // ...
};

class MyClass myClass;
MyClass myClass2;

class MyClass *pMyClass;
MyClass *pMyClass2;
```

```
class
{
    int field;
    // ...
} myOtherClass;

class
{
    int field;
    // ...
} *pMyOtherClass;
```



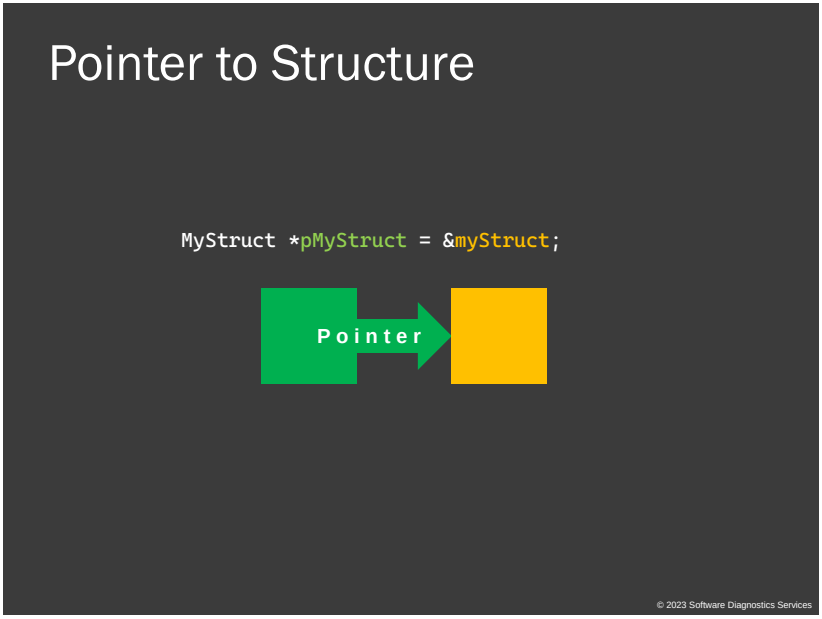
Both structures and classes are completely the same in C++ and can be used interchangeably. This is why you can always see `struct` in good modern C++ books. The only difference (if we ignore inheritance for now) is the field access, which is public by default in structures and private in classes.

The code example:

```
struct tagStruct
{
// public:
     int field;
     // ...
};
```

```
class tagClass
{
public: // (private:)
    int field;
    // ...
};
```

Pointer to Structure

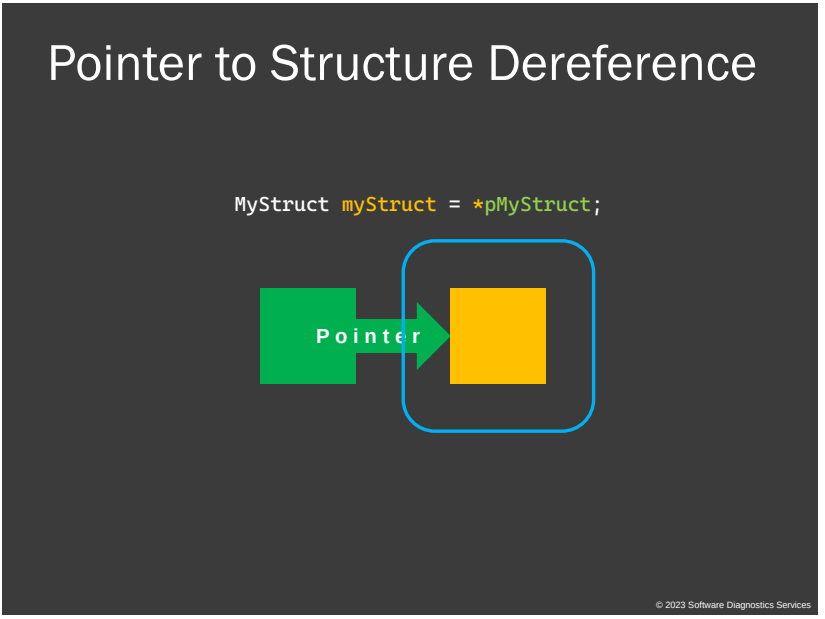


Here, we go again through our conceptual philosophy of pointers pictures and annotate them with C and C++ code.

The code example:

```
MyStruct *pMyStruct = &myStruct;
```

Pointer to Structure Dereference

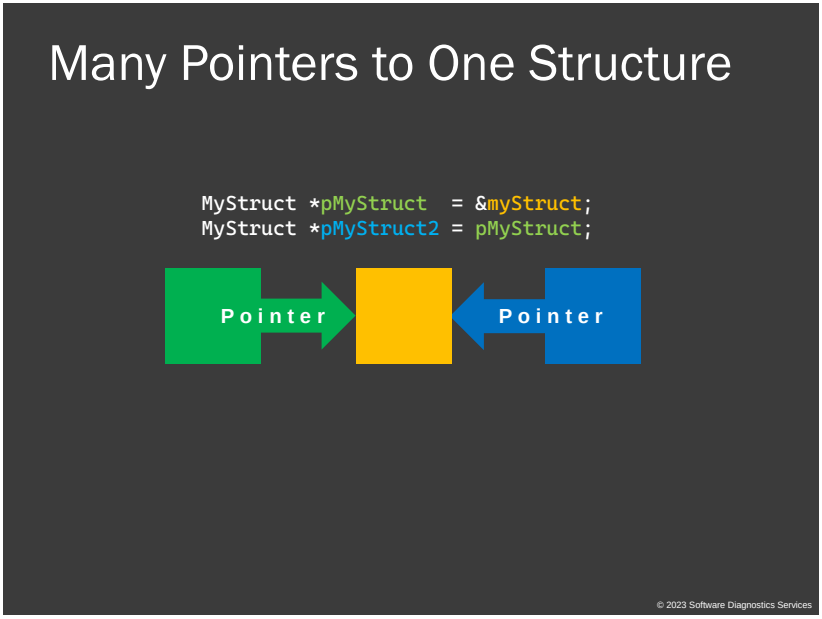


Here, we dereference a pointer to some structure. We get the structure value and copy-assign it to the `myStruct` variable.

The code example:

```
MyStruct myStruct = *pMyStruct;
```

Many Pointers to One Structure

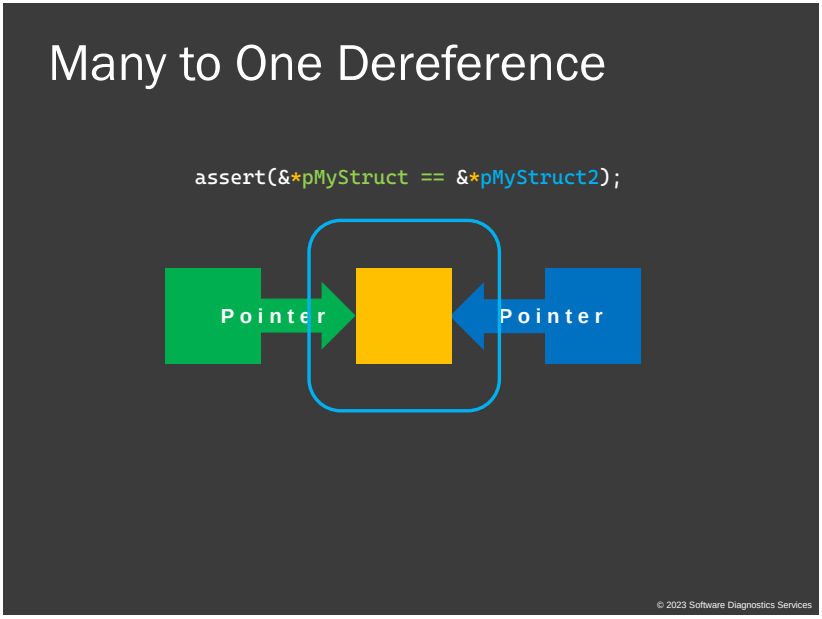


Here, we assign the value of one pointer to another, and both now point to the same structure.

The code example:

```
MyStruct *pMyStruct  = &myStruct;
MyStruct *pMyStruct2 = pMyStruct;
```

Many to One Dereference

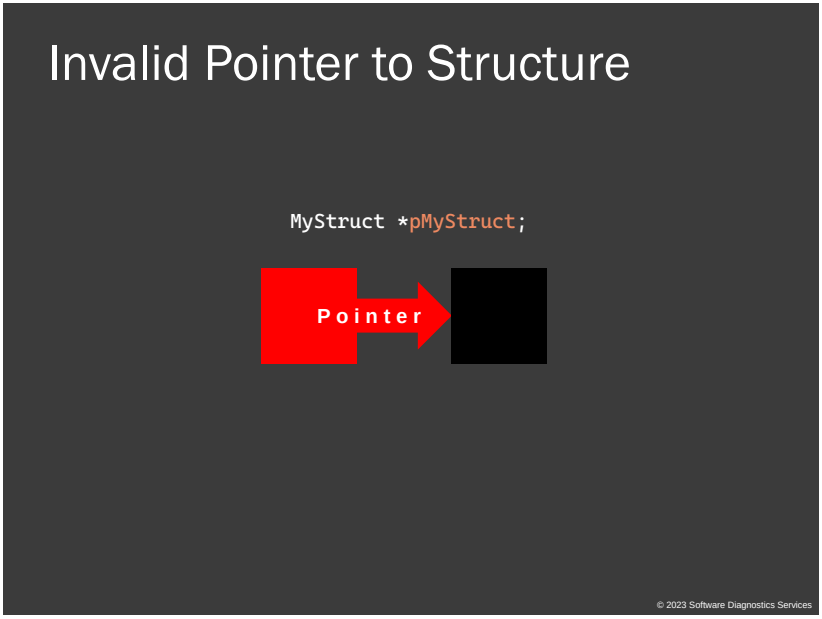


If we dereference both, we get the same value with the same address.

The code example:

```
assert(&*pMyStruct == &*pMyStruct2);
```

Invalid Pointer to Structure

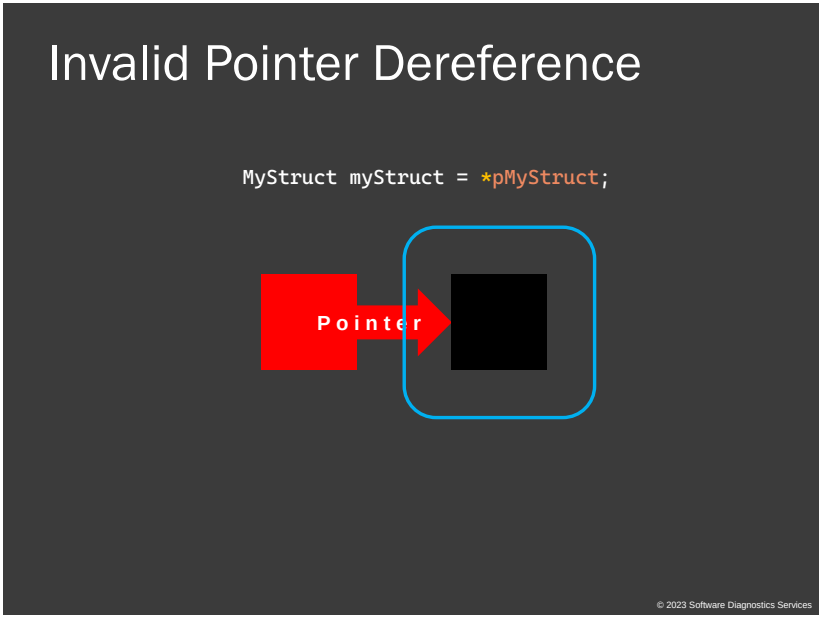


Here, we depict an uninitialized pointer that, depending on the memory storage type, can be a NULL pointer or some random value.

The code example:

```
MyStruct *pMyStruct;
```

Invalid Pointer Dereference

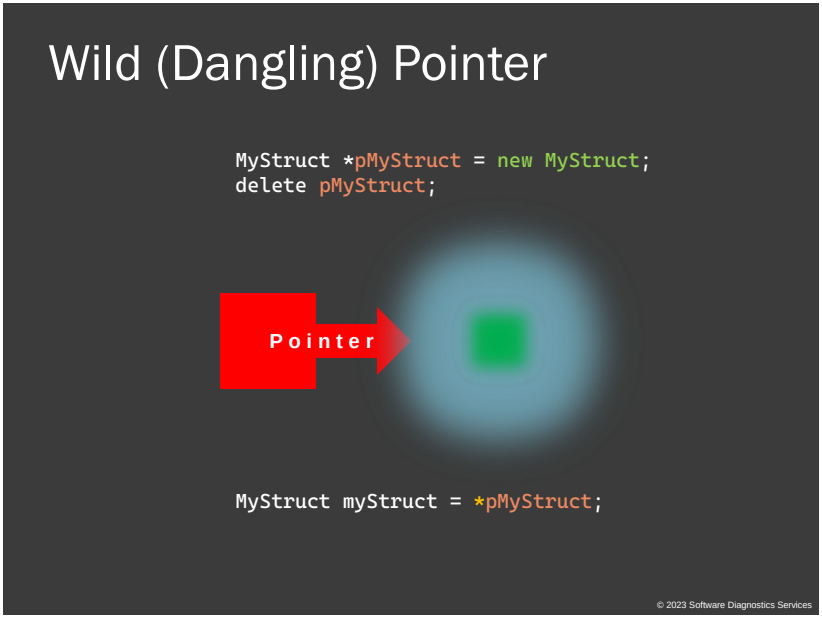


Dereferencing an uninitialized pointer can have undefined behavior, most likely an access violation leading to a crash.

The code example:

```
MyStruct *pMyStruct;
MyStruct myStruct = *pMyStruct;
```

Wild (Dangling) Pointer

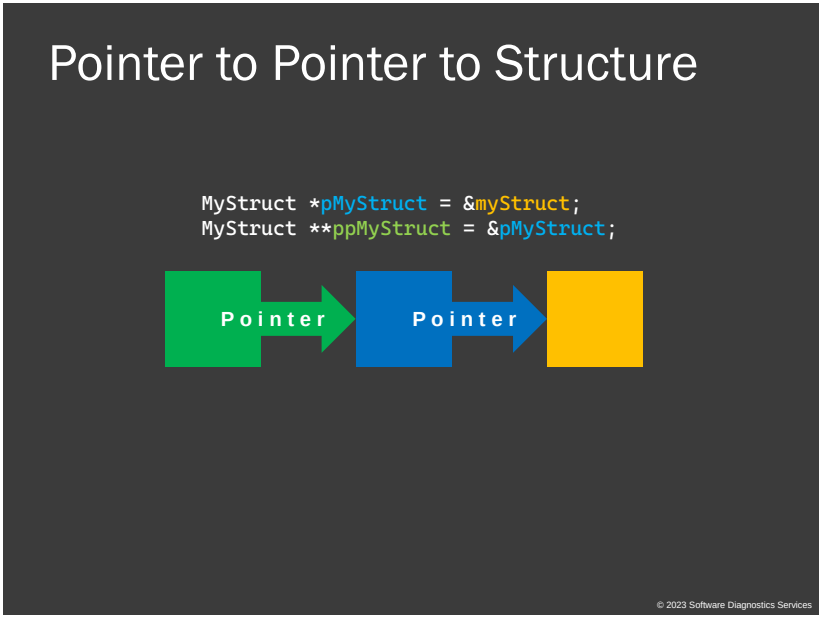


Memory for a structure can be dynamically allocated and then deallocated, but if a pointer is not reset to some value easy to check, such as 0, then we have a dangling pointer with its dereferencing resulting in undefined behavior that could lead to further corruption.

The code example:

```
MyStruct *pMyStruct = new MyStruct;
delete pMyStruct;

MyStruct myStruct = *pMyStruct;
```

Pointer to Pointer to Structure

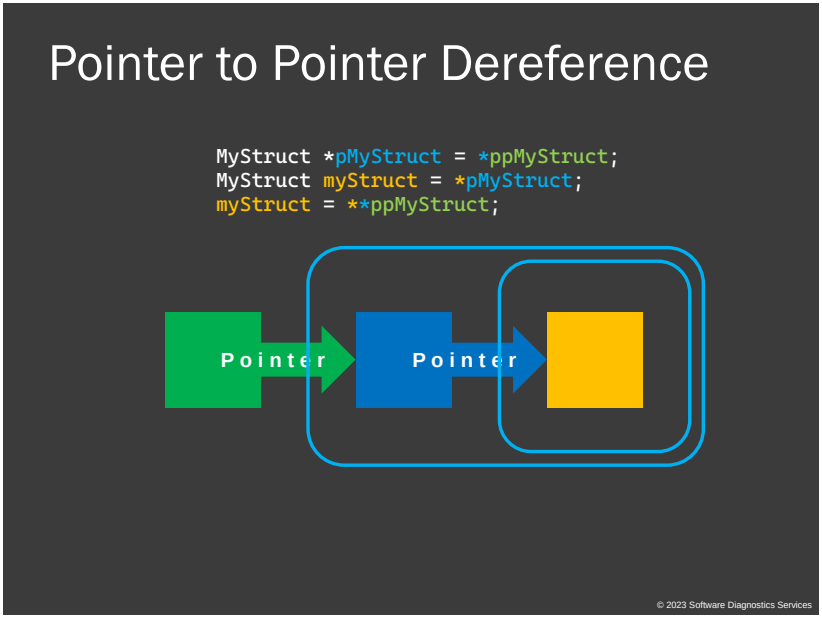


We can also have pointers to pointers to structures and so on, with double and more dereferences needed to get the value. We'll see why we need double-pointers later when we discuss passing parameters to functions.

The code example:

```
MyStruct *pMyStruct = &myStruct;
MyStruct **ppMyStruct = &pMyStruct;
```

Pointer to Pointer Dereference

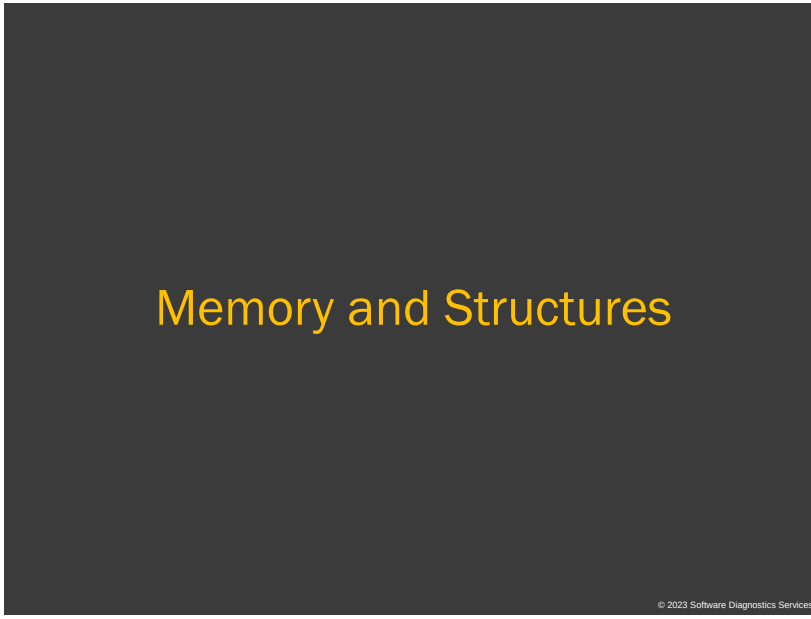


Here, we have double dereference illustrated.

The code example:

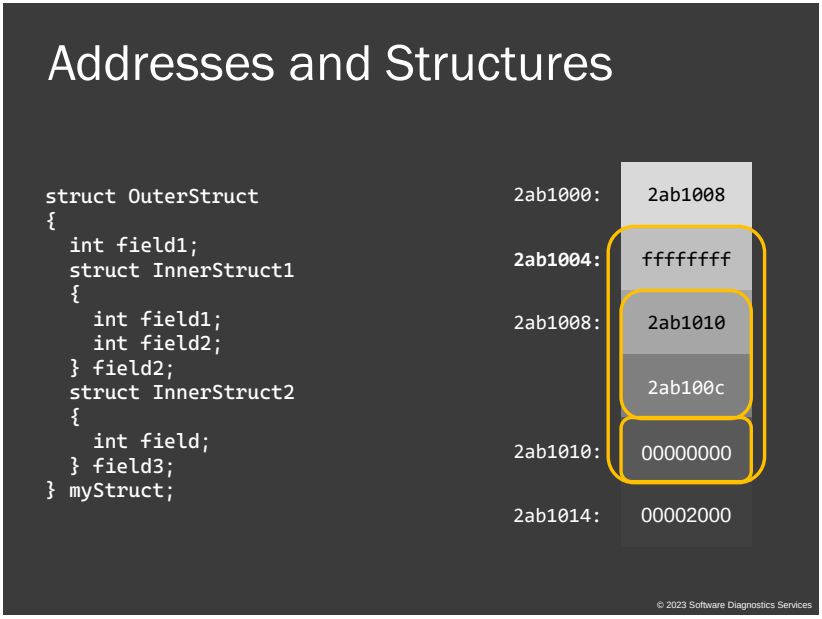
```
MyStruct *pMyStruct = *ppMyStruct;
MyStruct myStruct = *pMyStruct;
myStruct = **ppMyStruct;
```

Memory and Structures



Now, we look at the memory representation of structures.

Addresses and Structures

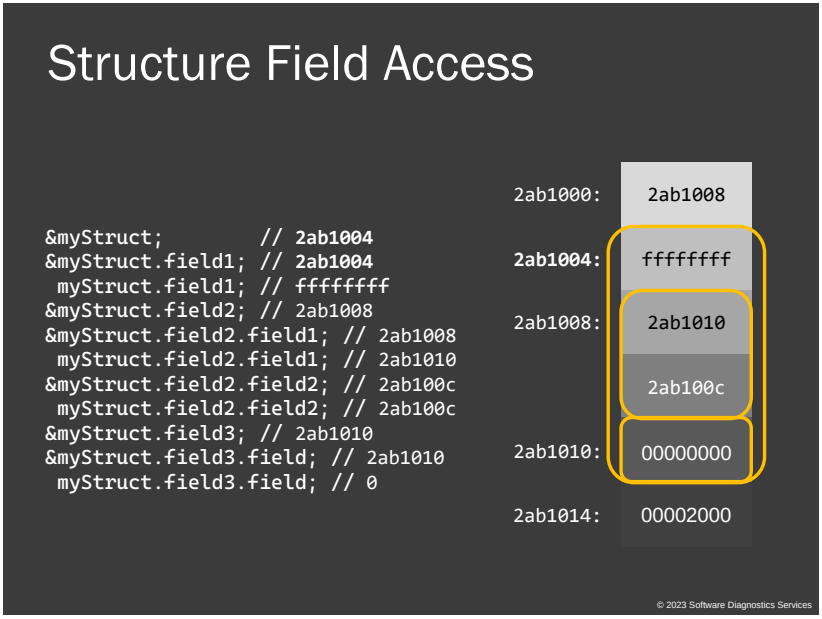


A structure in memory is a sequential collection of memory cells; some may be multicell and themselves substructures. Each part of a structure, its member, or structure field has its own address as well, in addition to the overall address of the structure.

The code example corresponding to the memory diagram:

```
struct OuterStruct
{
  int field1;
  struct InnerStruct1
  {
    int field1;
    int field2;
  } field2;
  struct InnerStruct2
  {
    int field;
  } field3;
} myStruct;
```

Structure Field Access

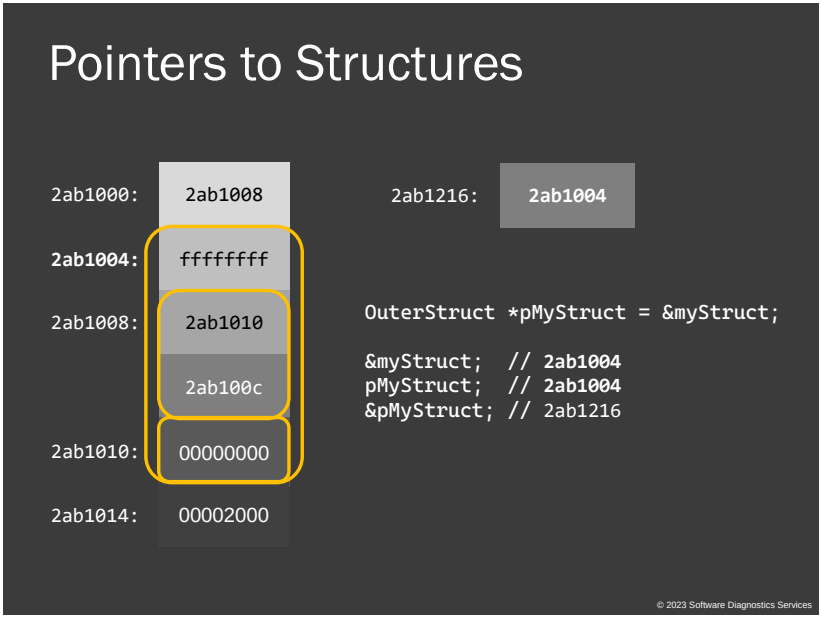


This example shows field addresses and access when we have a structure value.

The code example corresponding to the memory diagram:

```
&myStruct;         // 2ab1004
&myStruct.field1; // 2ab1004
 myStruct.field1; // ffffffff
&myStruct.field2; // 2ab1008
&myStruct.field2.field1; // 2ab1008
 myStruct.field2.field1; // 2ab1010
&myStruct.field2.field2; // 2ab100c
 myStruct.field2.field2; // 2ab100c
&myStruct.field3; // 2ab1010
&myStruct.field3.field; // 2ab1010
 myStruct.field3.field; // 0
```

Pointers to Structures

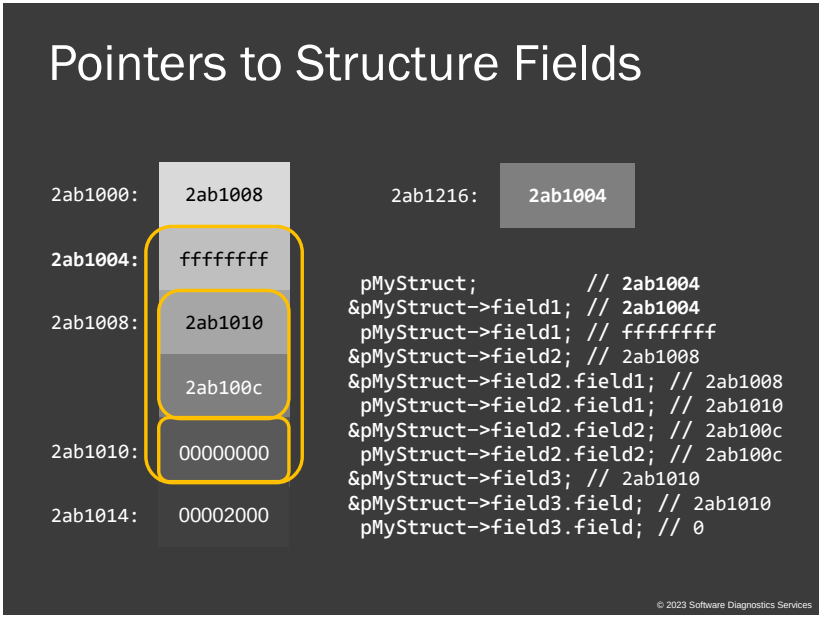


A structure has its address. A pointer to a structure is a memory cell that contains that address. It has its own address.

The code example corresponding to the memory diagram:

```
OuterStruct *pMyStruct = &myStruct;
&myStruct;  // 2ab1004
pMyStruct;  // 2ab1004
&pMyStruct; // 2ab1216
```

Pointers to Structure Fields

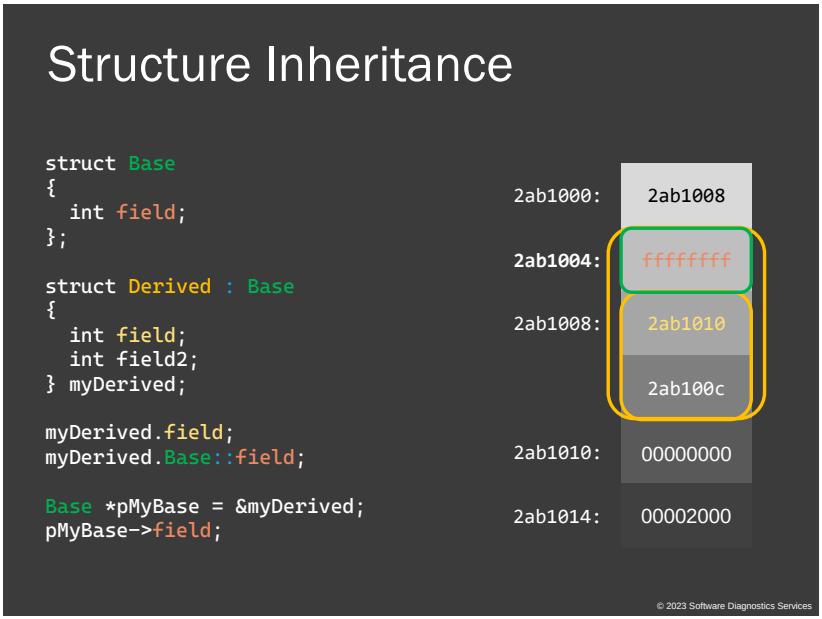


This example shows field addresses and access when we have a pointer to a structure value.

The code example corresponding to the memory diagram:

```
pMyStruct;           // 2ab1004
&pMyStruct->field1; // 2ab1004
 pMyStruct->field1; // ffffffff
&pMyStruct->field2; // 2ab1008
&pMyStruct->field2.field1; // 2ab1008
 pMyStruct->field2.field1; // 2ab1010
&pMyStruct->field2.field2; // 2ab100c
 pMyStruct->field2.field2; // 2ab100c
&pMyStruct->field3; // 2ab1010
&pMyStruct->field3.field; // 2ab1010
 pMyStruct->field3.field; // 0
```

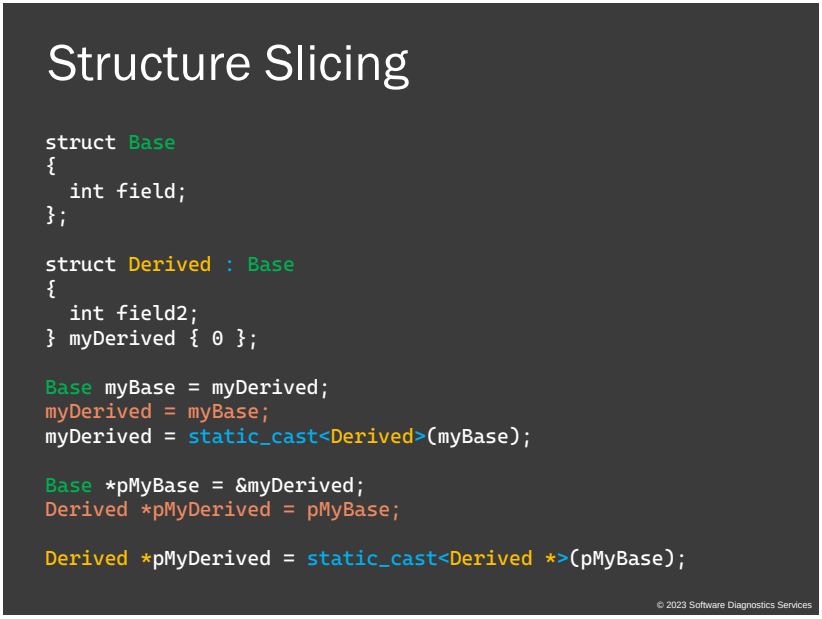
Structure Inheritance



Structures can inherit fields from other structures. In case of the same field names, the derived structure hides the base structure fields, but they can be accessed by explicit base structure name qualification:

```
struct Base
{
  int field;
};

struct Derived : Base
{
  int field;
  int field2;
} myDerived;

myDerived.field;
myDerived.Base::field;

Base *pMyBase = &myDerived;
pMyBase->field;
```

Structure Slicing



It is possible to copy a derived structure to a base structure variable, but in this case, the former contents are sliced since the base structure occupies less memory. The other way around, from the base structure to the derived, is forbidden by default because the compiler doesn't know how to fill the new derived-only fields. However, this can be forced with a static cast where the derived fields are filled with the existing adjacent memory content, which can be completely random. The same downcast can be done between pointers, but when we try to dereference a target pointer to the derived structure later, we may get random data.

The code example:

```
struct Base
{
  int field;
};
```

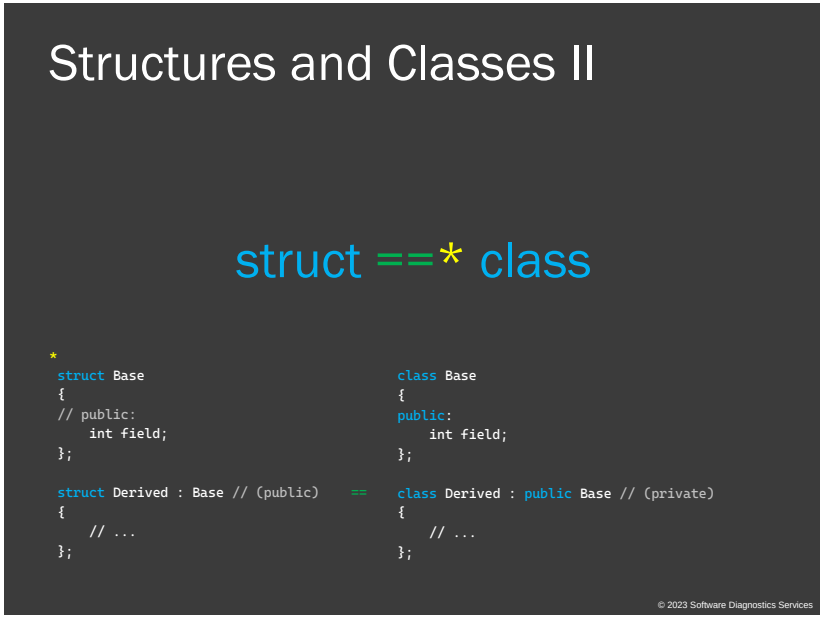
```
struct Derived : Base
{
  int field2;
} myDerived { 0 };

Base myBase = myDerived;
myDerived = myBase;
myDerived = static_cast<Derived>(myBase);

Base *pMyBase = &myDerived;
Derived *pMyDerived = pMyBase;
Derived *pMyDerived = static_cast<Derived *>(pMyBase);
```

Inheritance Access Level

It is possible to inherit privately. In such a case, the base structure fields are inaccessible from the outside, even with the explicit qualification.

The code example:

```
struct Base
{
  int field;
};

struct Derived : private Base
{
  int field;
  int field2;
} myDerived;

myDerived.field;
myDerived.Base::field;

Base *pMyBase = &myDerived;
pMyBase->field;
```



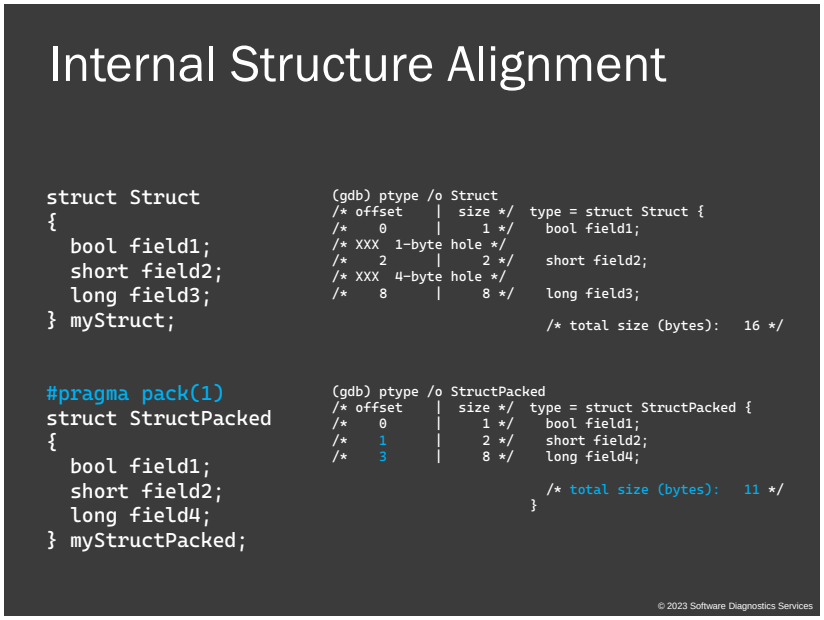
Again, structures and classes are almost equivalent except for the default inheritance access (and field access), which is, by default, public for structures and private for classes. Public access needs to be specified explicitly for classes. We do not discuss protected access in this training, which is not really relevant for memory thinking when looking at built code.

The code example:

```
struct Base
{
// public:
    int field;
};


struct Derived : Base
// (public)
{
    // ...
};
```

```
class Base
{
public:
    int field;
};


class Derived : public Base
// (private)
{
    // ...
};
```

Internal Structure Alignment

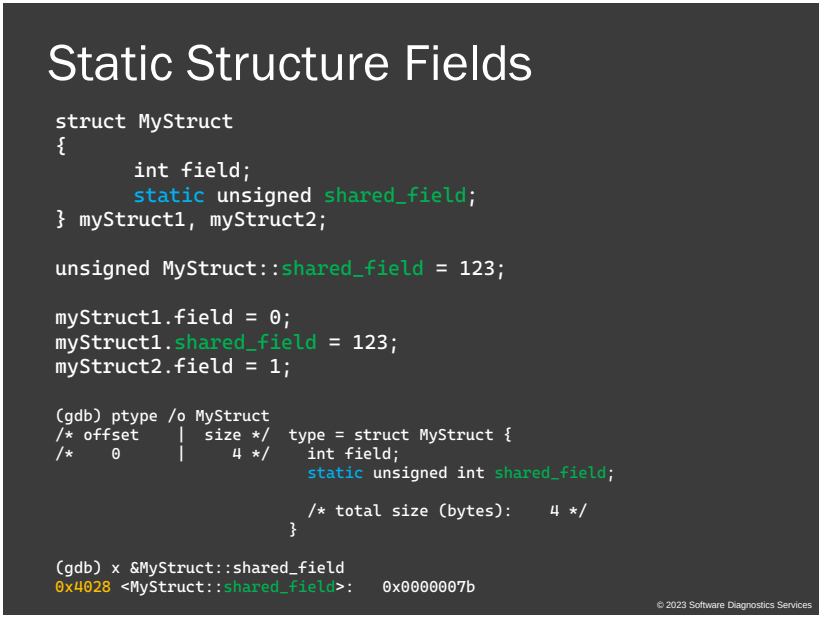


Fields may be aligned according to their default type alignment, which may introduce gaps, increasing the overall structure size.

The code example corresponding to the GDB output:

```
struct Struct
{
  bool field1;
  short field2;
  long field3;
} myStruct;

#pragma pack(1)
struct StructPacked
{
  bool field1;
  short field2;
  long field4;
} myStructPacked;
```

Static Structure Fields



Static structure field values are shared between the different objects of the same structure type. They occupy uniquely separate memory cells from the objects' memory.

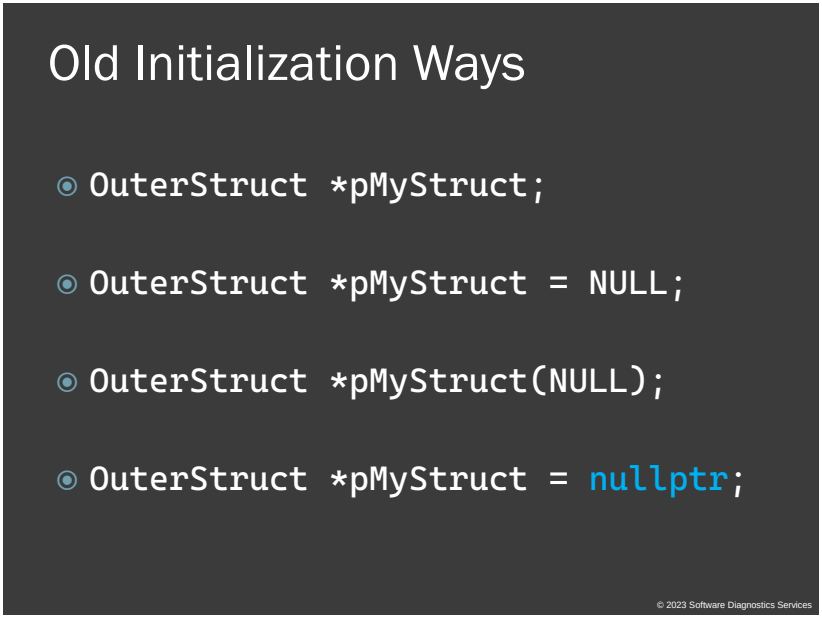
The code example corresponding to the GDB output:

```
struct MyStruct
{
      int field;
      static unsigned shared_field;
} myStruct1, myStruct2;

unsigned MyStruct::shared_field = 123;

myStruct1.field = 0;
myStruct1.shared_field = 123;
myStruct2.field = 1;
```

Uniform Initialization

Throughout C++ history, there were several ways to initialize variables. Finally, there is some uniform way to do it consistently.

Old Initialization Ways

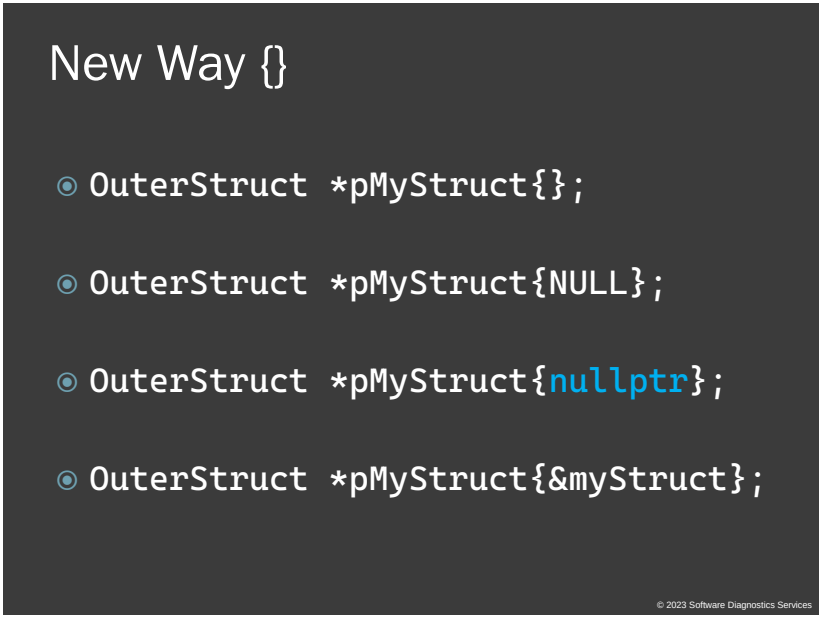


When we omit an initialization value, a variable is considered uninitialized if its memory belongs to certain memory classes, such as stack. For static memory, it may be default-initialized with zero memory values.

The code example:

```
OuterStruct *pMyStruct;
OuterStruct *pMyStruct = NULL;
OuterStruct *pMyStruct(NULL);
OuterStruct *pMyStruct = nullptr;
```

New Way {}

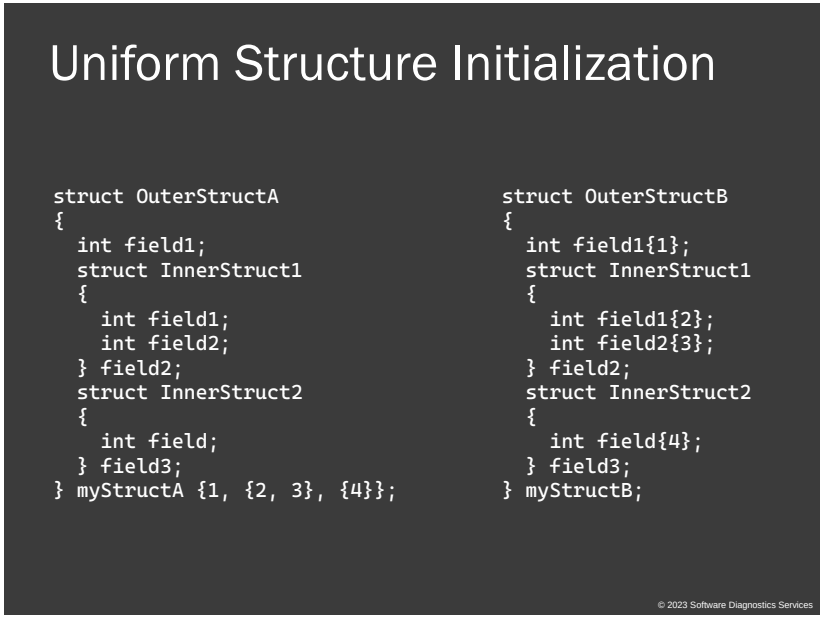


When we use the new way of initialization in modern C++, we can use empty `{}` to signal default initialization even for stack memory.

The code example:

```
OuterStruct *pMyStruct{};
OuterStruct *pMyStruct{NULL};
OuterStruct *pMyStruct{nullptr};
OuterStruct *pMyStruct{&myStruct};
```

Uniform Structure Initialization



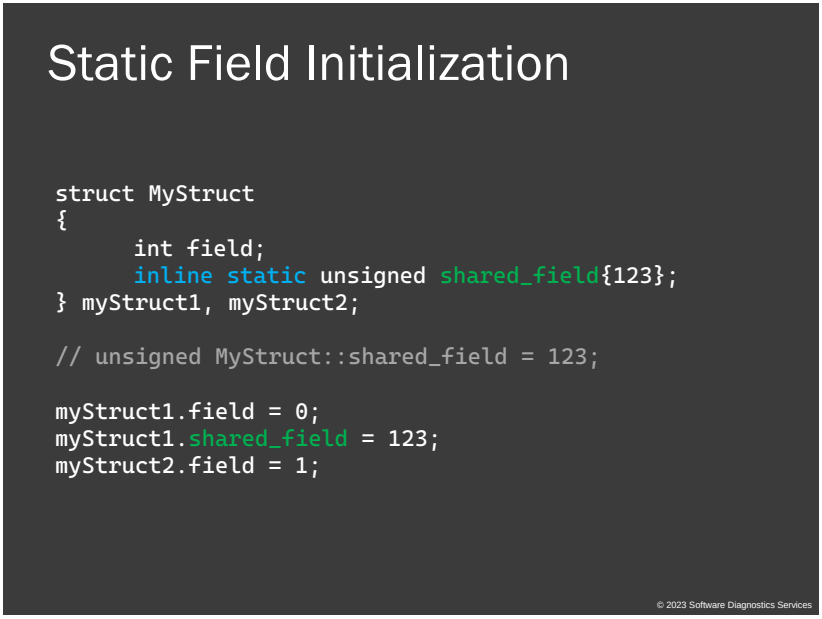
It is possible to uniformly initialize the structure outside or provide default field initializers in the structure definition.

The code example:

```
struct OuterStructA
{
  int field1;
  struct InnerStruct1
  {
    int field1;
    int field2;
  } field2;
  struct InnerStruct2
  {
    int field;
  } field3;
} myStructA {1, {2, 3}, {4}};
```

```
struct OuterStructB
{
  int field1{1};
  struct InnerStruct1
  {
    int field1{2};
    int field2{3};
  } field2;
  struct InnerStruct2
  {
    int field{4};
  } field3;
} myStructB;
```

Static Field Initialization

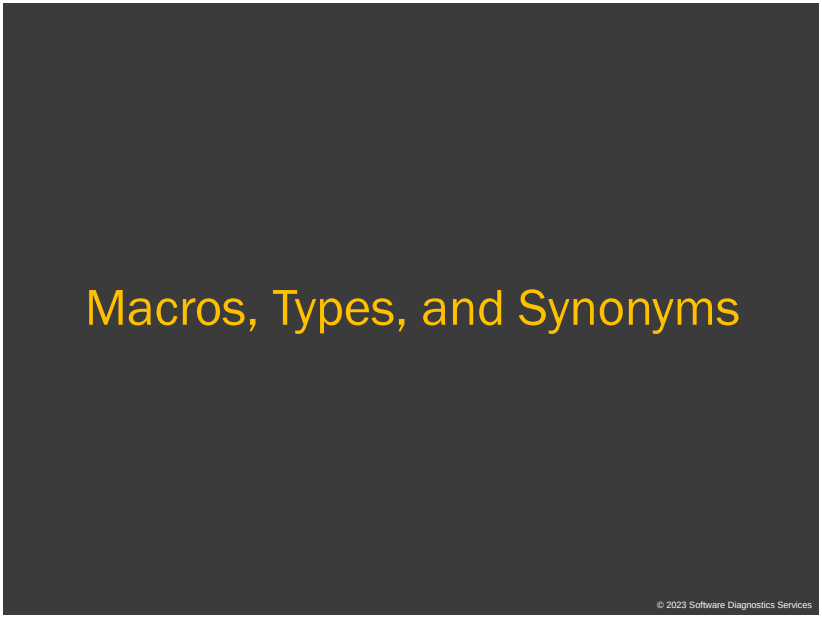


The latest C++ standards allow static field initialization inside the structure definition instead of the classic C++ ways of outside initialization (shown in comments).

The code example:

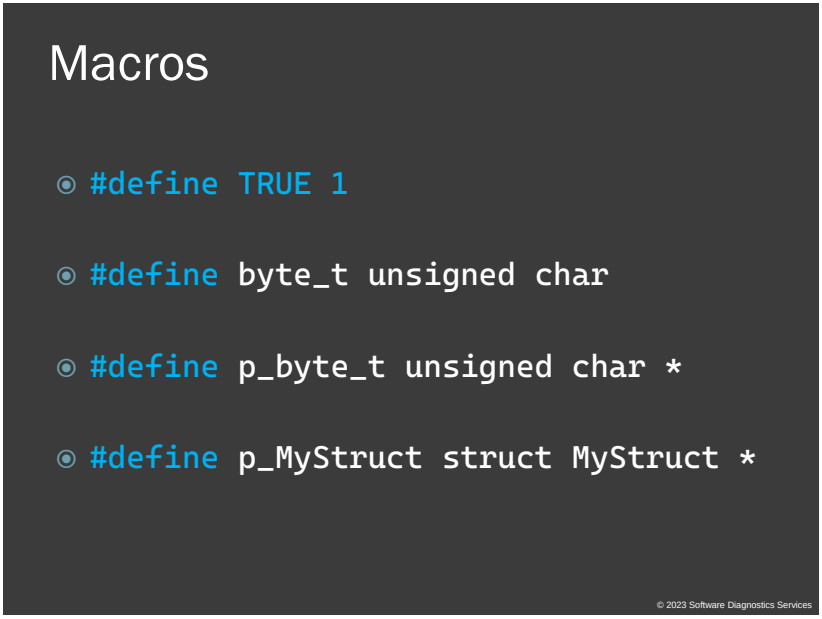
```
struct MyStruct
{
      int field;
      inline static unsigned shared_field{123};
} myStruct1, myStruct2;

// unsigned MyStruct::shared_field = 123;

myStruct1.field = 0;
myStruct1.shared_field = 123;
myStruct2.field = 1;
```

Macros, Types, and Synonyms



Type names may be long or inconvenient. There are some ways to construct easier type names.

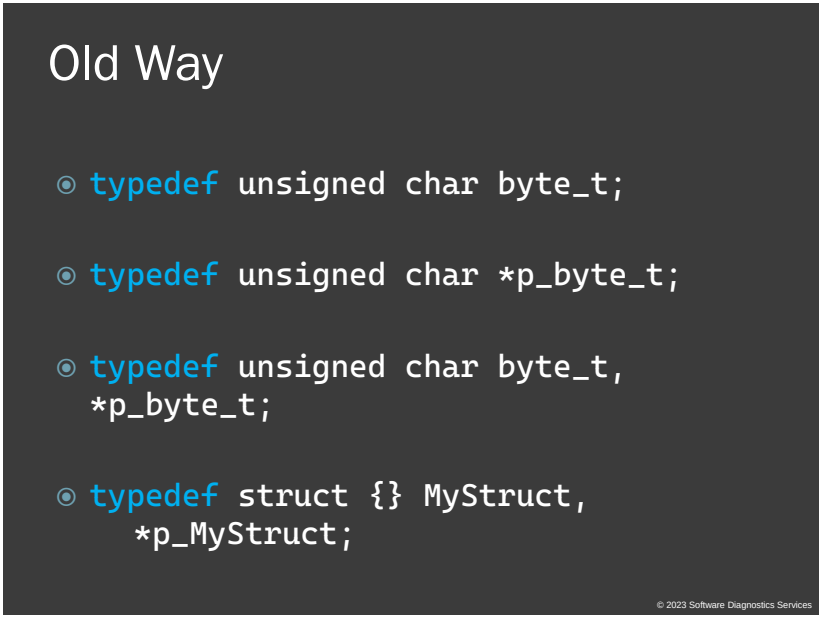
Macros



The code example:

```
#define TRUE 1
#define byte_t unsigned char
#define p_byte_t unsigned char *
#define p_MyStruct struct MyStruct *
```

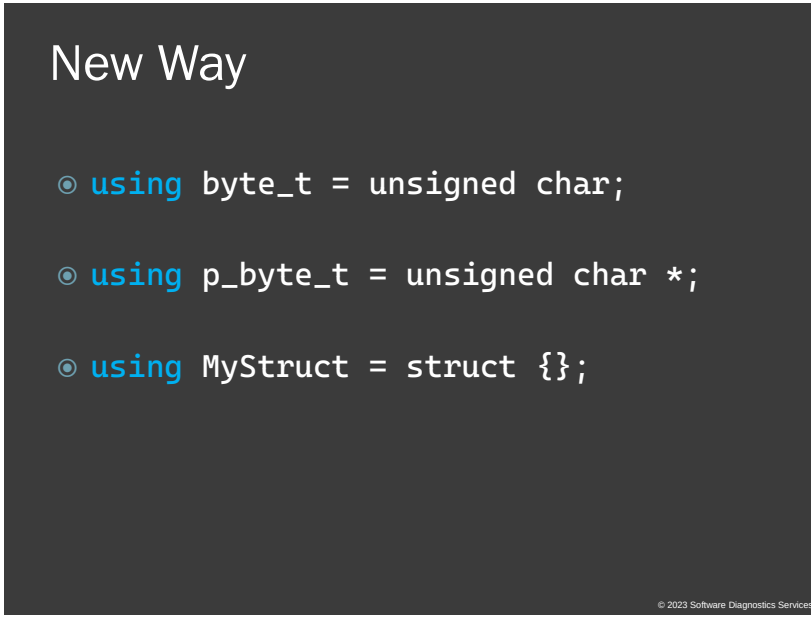
Old Way



The code example:

```
typedef unsigned char byte_t;
typedef unsigned char *p_byte_t;
typedef unsigned char byte_t, *p_byte_t;
typedef struct {} MyStruct, *p_MyStruct;
```

New Way



The code example:

```
using byte_t = unsigned char;
using p_byte_t = unsigned char *;
using MyStruct = struct {};
```

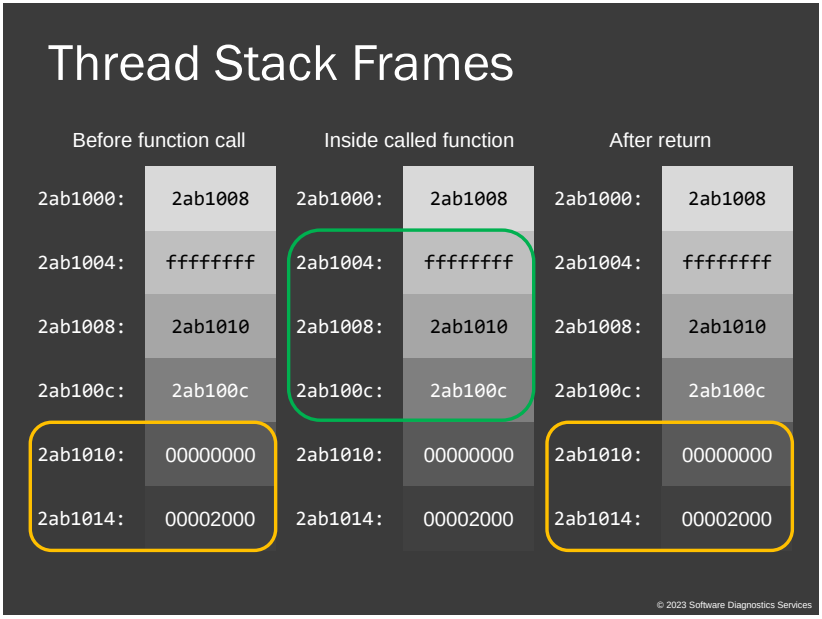
Memory Storage

What memory storage is used to store values ultimately influences program behavior and possible defects.

Overview

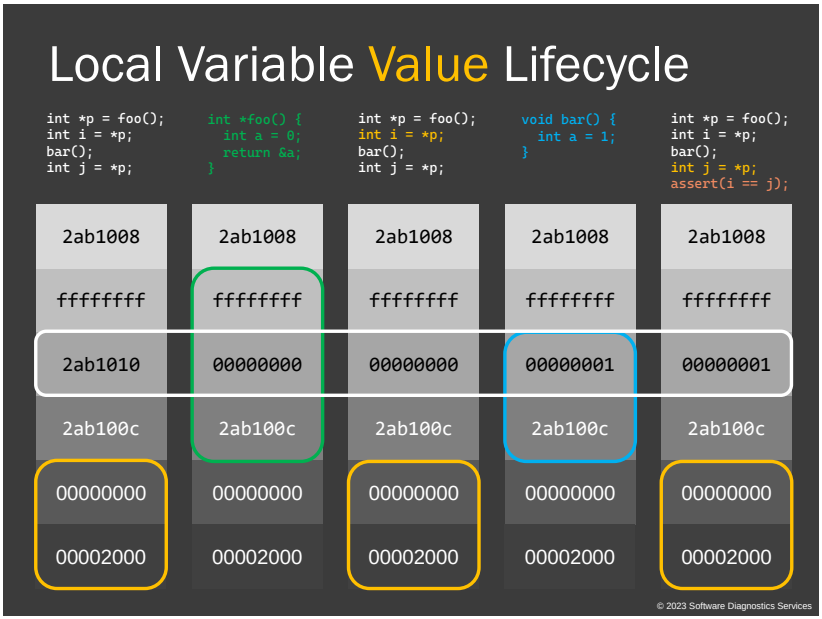
Here, we show the list of different storage types and talk about them in detail later. We cover the polymorphic allocators in the next edition.

Thread Stack Frames



When a function is called, a stack frame is allocated in the thread stack memory region to hold local variables' values.

Local Variable Value Lifecycle



Since the stack frame memory values can be overwritten after the return from the function by subsequent function calls, local variable values have definite values only during the function call where they were initialized.

The code examples corresponding to the memory diagrams.

Before calling the `foo` function, the memory values below the current stack frame are undefined:

```
int *p = foo();
int i = *p;
bar();
int j = *p;
```

When we enter the `foo` function, the corresponding stack frame is created. The function code also initializes the local variable `a` with 0 value. The function also returns the stack address of that local variable:

```
int *foo()
{
  int a = 0;
  return &a;
}
```

In the caller, we save that value at that address in the `i` variable. Then we call the `bar` function:

```
int *p = foo();
int i = *p;
bar();
int j = *p;
```

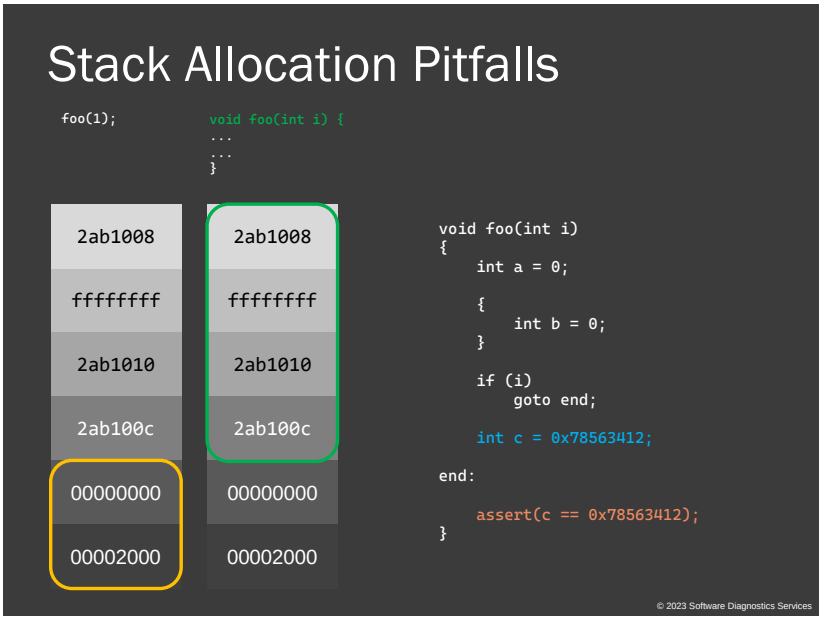
When we enter the `bar` function, the corresponding stack frame is created. The function code also initializes the local variable `a` with 1. Coincidentally, the variable `a` occupies the same stack memory location as the local variable `a` in the previous `foo` function call:

```
void bar()
{
  int a = 1;
}
```

Upon the return from the `bar` function, we dereference the same `p` address but get a different value:

```
int *p = foo();
int i = *p;
bar();
int j = *p;
assert(i == j);
```

Stack Allocation Pitfalls



Please don't forget that stack frame memory for all function local variables is allocated at the entrance of the function, but individual variables may be initialized at a later time.

The code example corresponding to the memory diagram:

```
foo(1);
```

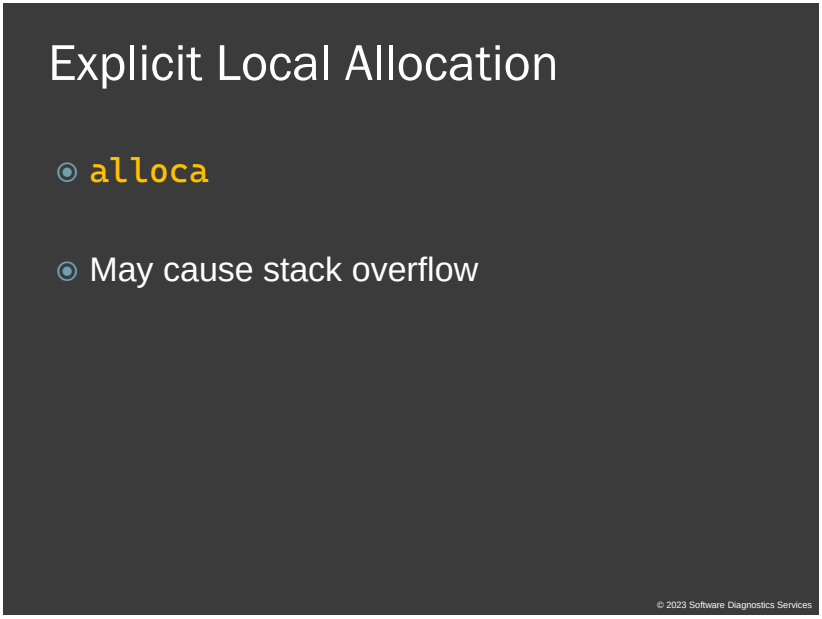
When we enter the `foo` function, the allocated stack frame includes the local variable `c`, which is initially uninitialized:

```
void foo(int i)
{
...
...
}
```

If the value of the `i` function parameter is positive, the initialization of the local variable `c` is skipped, and the assertion is failed:

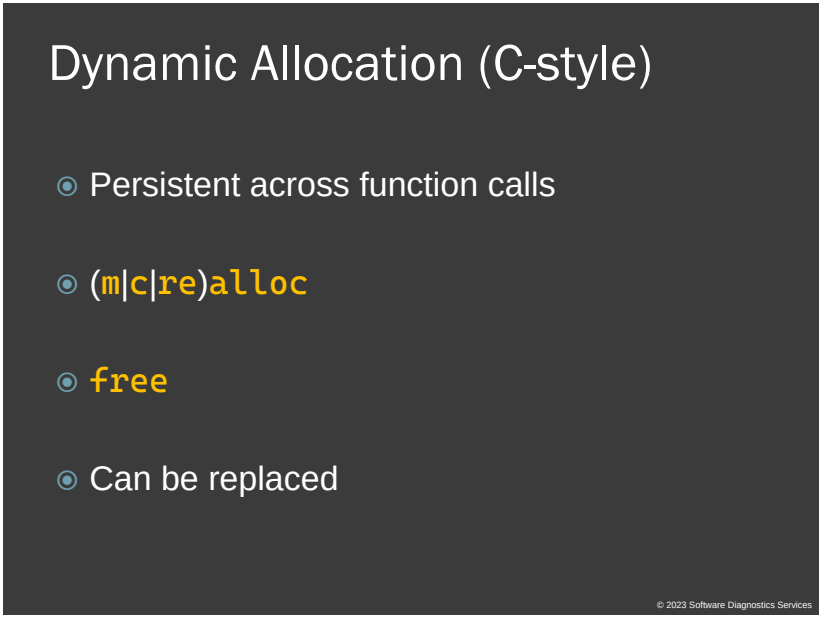
```
void foo(int i)
{
    int a = 0;
    {
        int b = 0;
    }
    if (i)
        goto end;
    int c = 0x78563412;
end:

    assert(c == 0x78563412);
}
```

Explicit Local Allocation



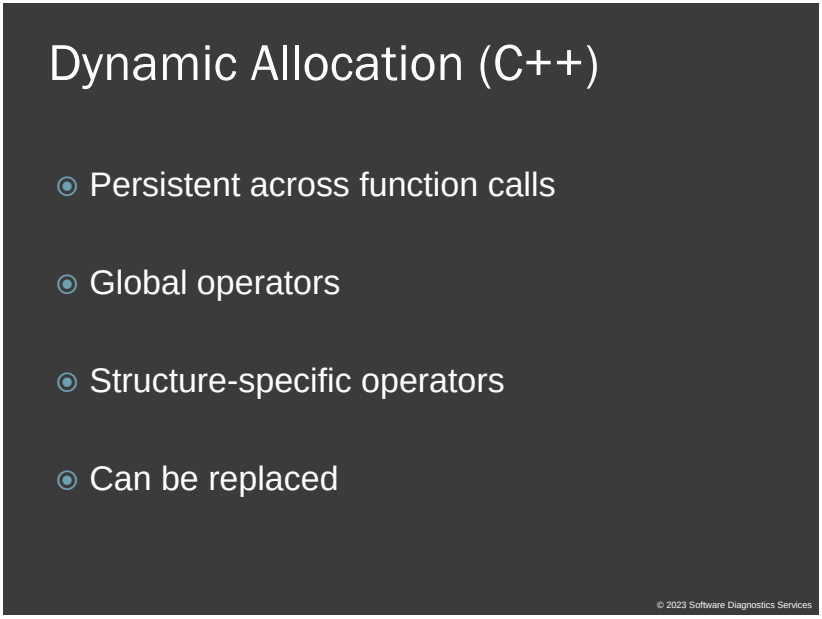
It is possible to explicitly allocate memory on the thread stack, for example, for some variable-length array storage. However, be aware of the possible stack overflow.

Dynamic Allocation (C-style)



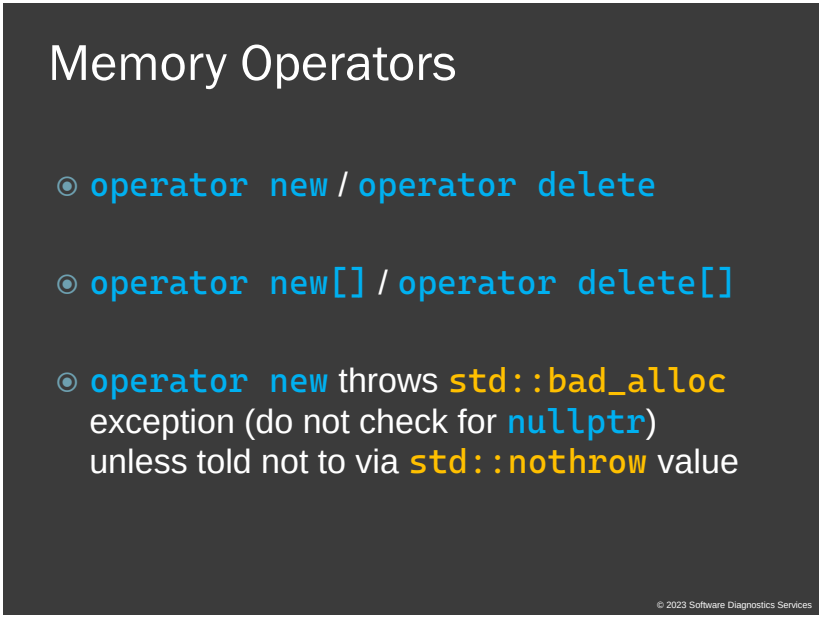
There are some advantages to a dynamic memory allocation compared to a local stack allocation. The allocated memory and its values persist across function calls. Since allocations are implemented by library calls, they can be replaced with other libraries and custom code that provides debugging capabilities for tracking memory allocations and deallocations, as well as other checks.

Dynamic Allocation (C++)



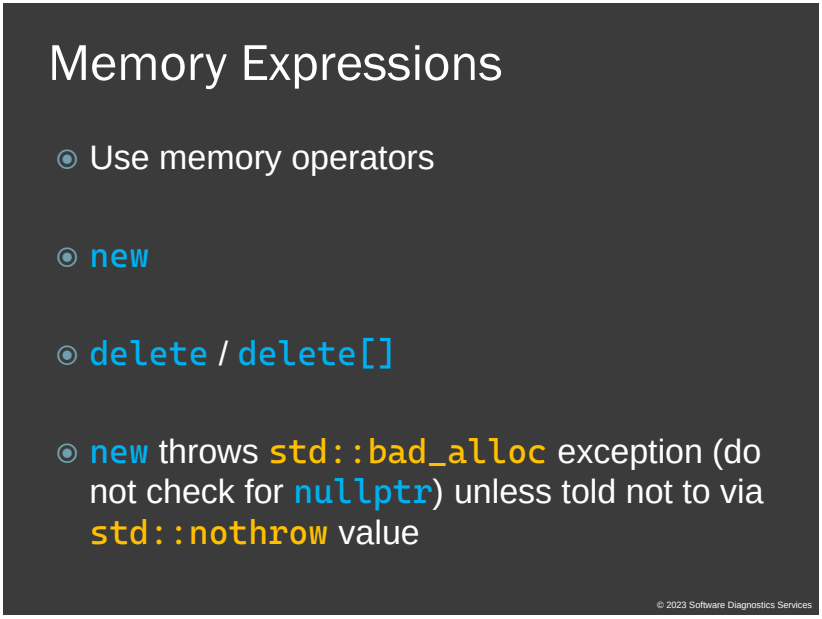
C++ has its own implementation of dynamic memory that is often internally implemented by underlying C-style calls and Linux API. However, these high-level allocation facilities are more flexible and customizable to the needs of structure designers. It provides replaceable operators for global allocations for chunks of memory and structure-specific allocations.

Memory Operators



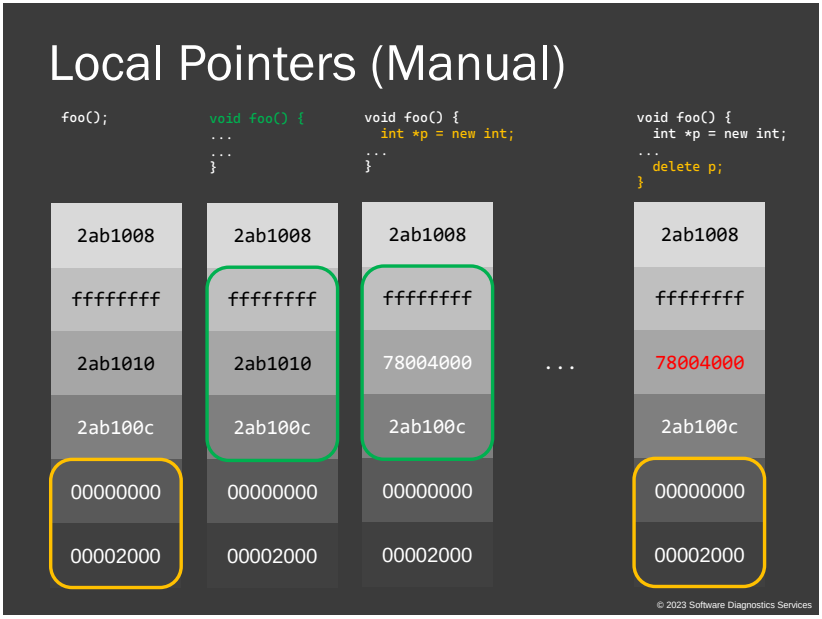
When freeing globally allocated memory, always pay attention to whether it was allocated in the array form to avoid memory leaks, crashes, and other undefined behavior. Also, never check the allocated memory address for `nullptr` as done in the C-style allocations: C++ allocation operators throw an exception instead.

Memory Expressions



Memory allocation expressions are used for allocating memory for values, structures, and their arrays. Internally, they may use memory operators. The same advice for non-array/array deallocation and checking return addresses is applicable here.

Local Pointers (Manual)



When allocating memory dynamically and assigning the memory address to a local variable, we must not forget to free/delete memory before returning from the function to avoid a memory leak.

The code example corresponding to the memory diagram:

```
foo();
```

When we enter the `foo` function, the allocated stack frame includes the local variables:

```
void foo()
{
...
...
}
```

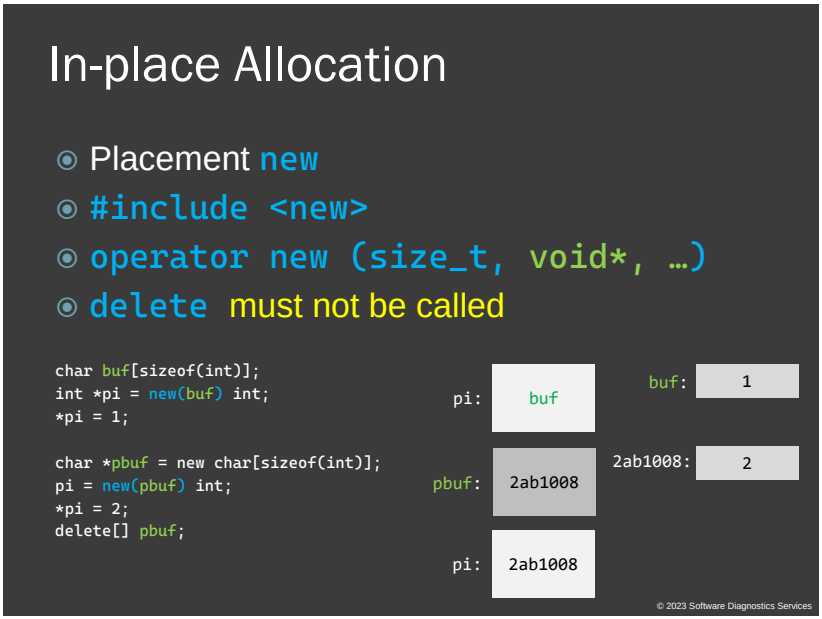
The local variable `p` contains the address of the allocated memory for an integer value:

```
void foo()
{
  int *p = new int;
...
}
```

However, before exiting the function, we must free the memory; otherwise, there is a memory leak. Please note that neither `delete` nor `free` change the value of the variable `p`. It becomes a dangling pointer but it is ok because it goes out of scope here and is not reused for dereferencing unless saved somewhere else.

```
void foo()
{
  int *p = new int;
...
  delete p;
}
```

In-place Allocation



If we want to reuse existing memory buffers, we can use placement `new`.

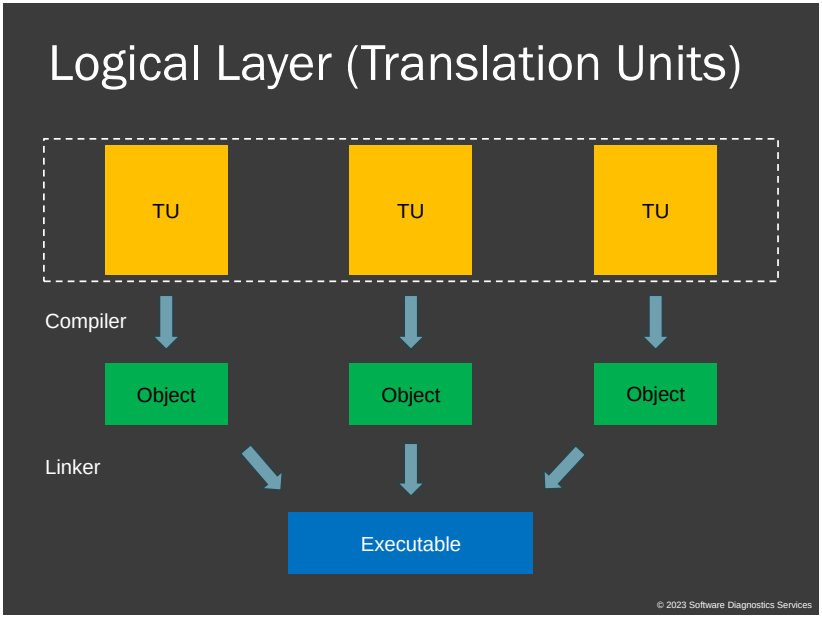
The code examples corresponding to the memory diagrams:

```
char buf[sizeof(int)];
int *pi = new(buf) int;
*pi = 1;

char *pbuf = new char[sizeof(int)];
pi = new(pbuf) int;
*pi = 2;
delete[] pbuf;
```

Source Code Organisation

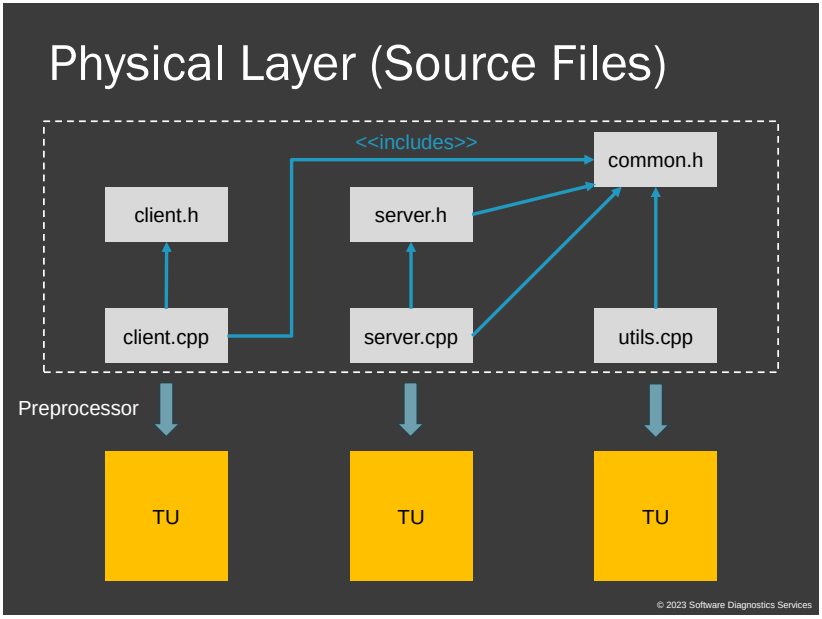
We now discuss C and C++ source code organization.

Logical Layer (Translation Units)



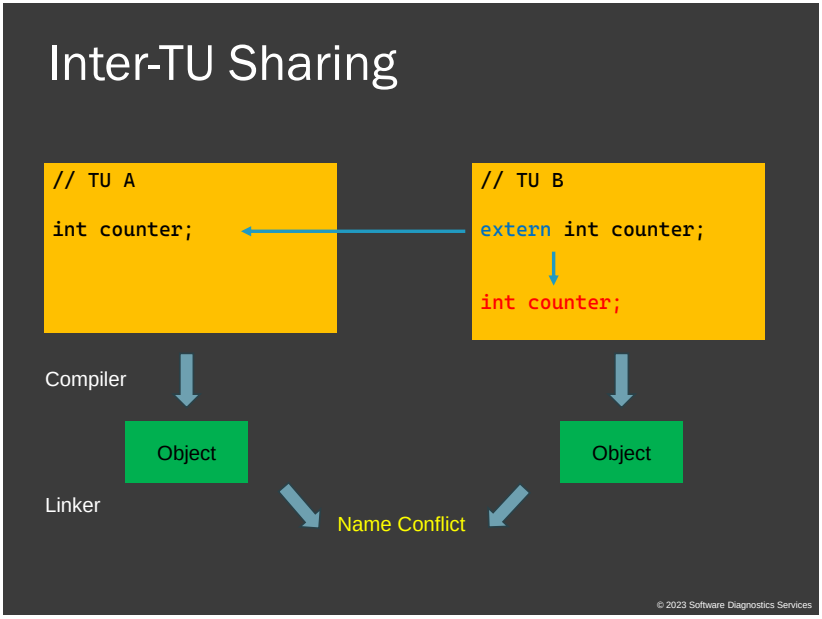
From a compiler perspective, it works with a translation unit as a whole and converts the source code of a translation unit to an object file. Several object files are combined by a linker into an executable file.

Physical Layer (Source Files)



Although one physical source code file corresponds to one translation unit, it is passed through a preprocessor, which, among other things, looks for special directives to include other files, and those files may also contain directives to include other files. You can also see, as in the case of the *common.h* file, by transitivity of inclusion, that the same file may be included many times.

Inter-TU Sharing



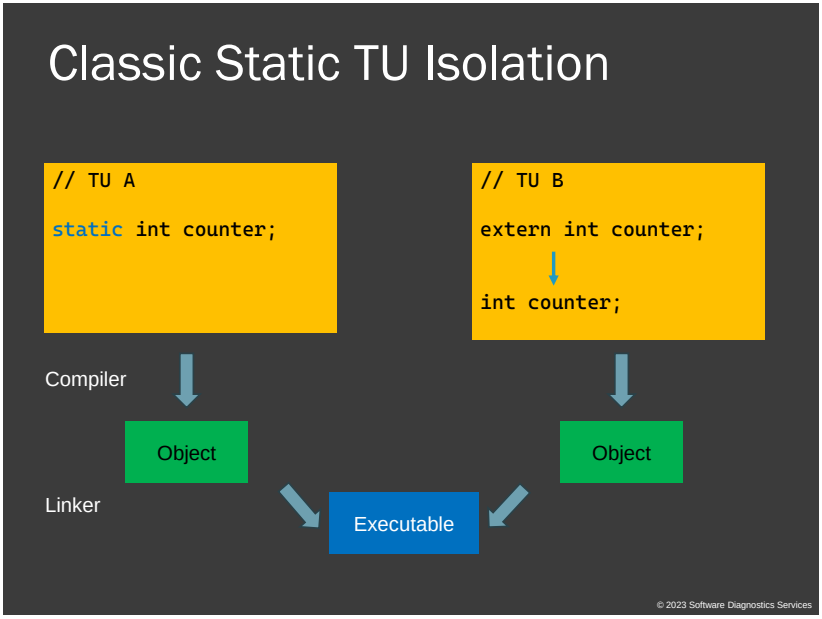
Variables in different translation units having the same name may conflict during the linkage phase.

The code examples corresponding to the diagrams:

```
// TU A

int counter;
```

```
// TU B

extern int counter;

int counter;
```

Classic Static TU Isolation



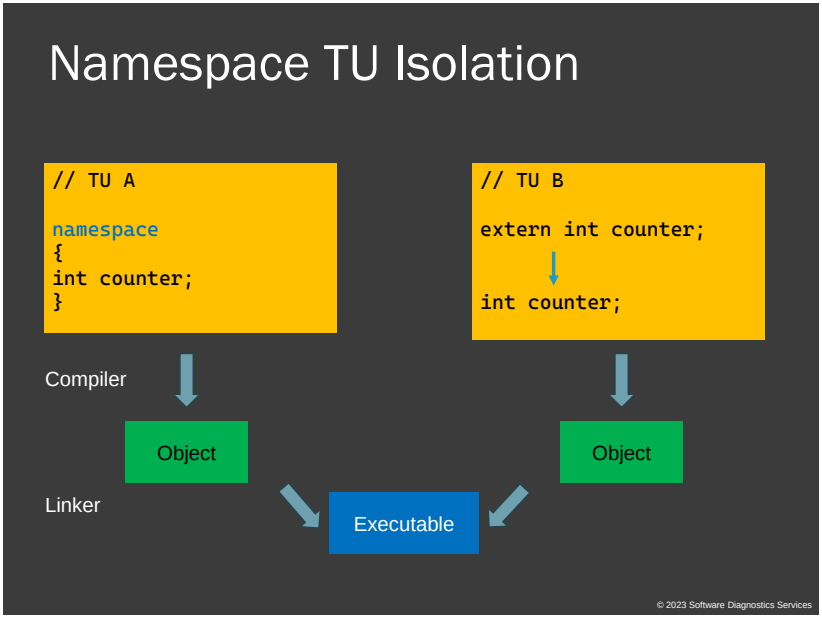
To avoid name conflicts during linkage, C and classic C++ suggest using the `static` specifier.

The code examples corresponding to the diagrams:

```
// TU A

static int counter;
```

```
// TU B

extern int counter;

int counter;
```

Namespace TU Isolation



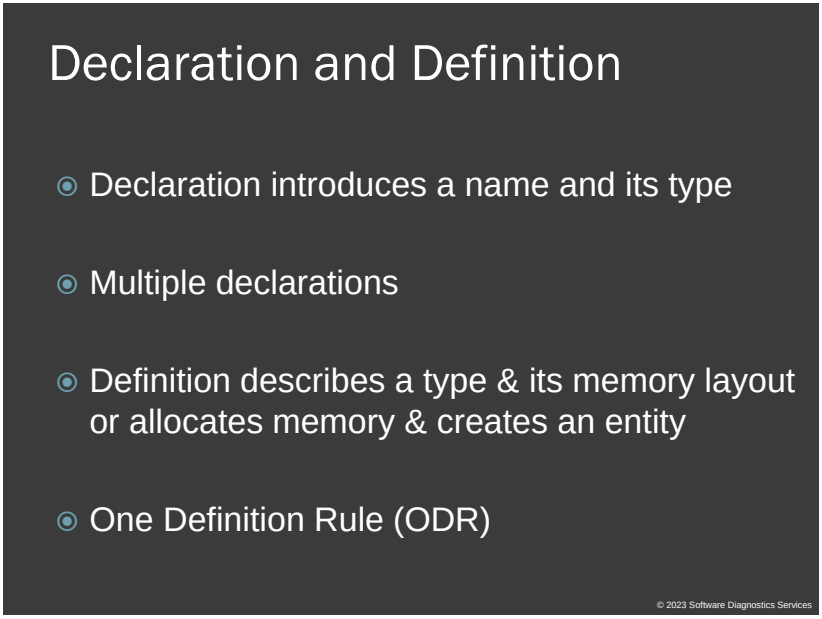
Modern C++ suggests using namespaces instead.

The code examples corresponding to the diagrams:

```
// TU A

namespace
{
int counter;
}
```

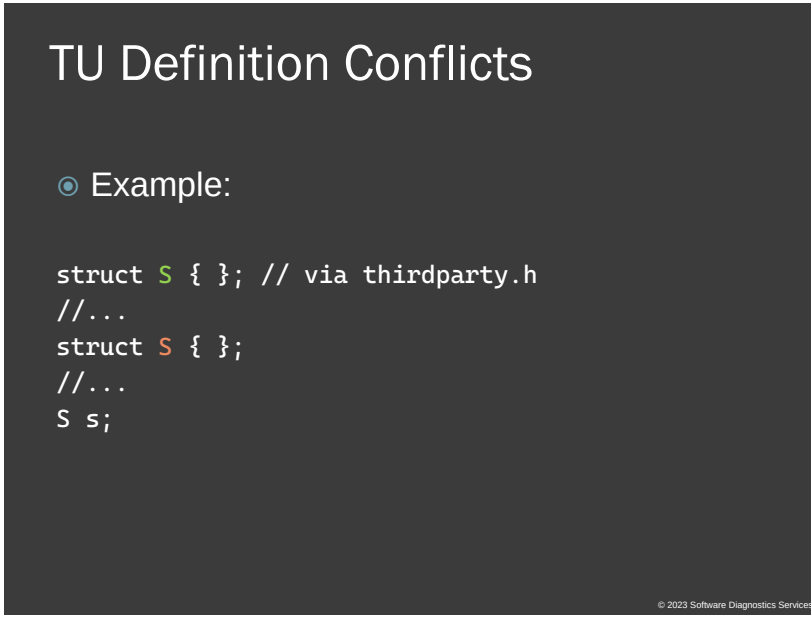
```
// TU B

extern int counter;

int counter;
```

Declaration and Definition



In C and C++, when reasoning about compilation, it is useful to consider the distinction between declaration and definition. The rule of thumb is that the latter usually describes the memory layout. Please also note that a definition is also a declaration.

TU Definition Conflicts

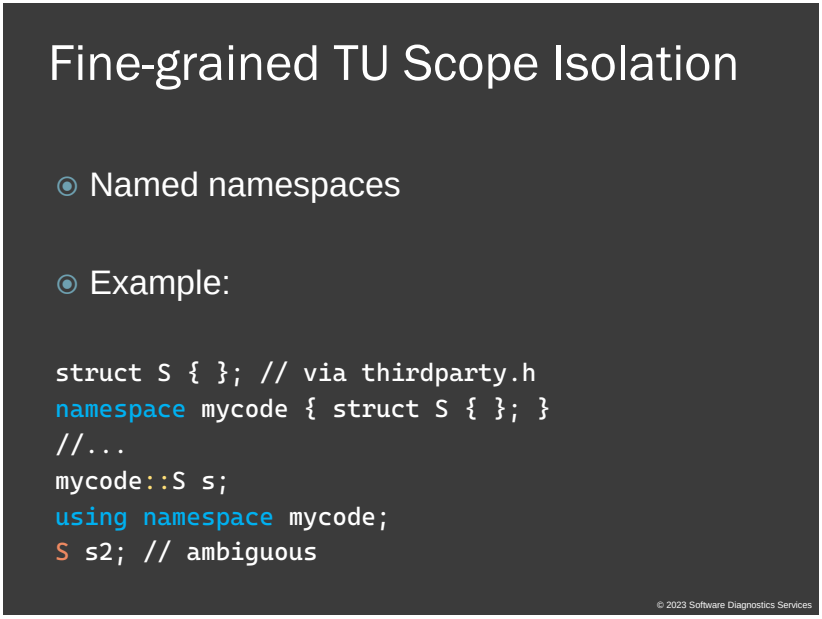


Multiple declarations of the same entity are allowed, but only one definition is allowed, the essence of ODR, One Definition Rule.

The code example:

```
struct S { }; // via thirdparty.h
//...
struct S { };
//...
S s;
```

Fine-grained TU Scope Isolation

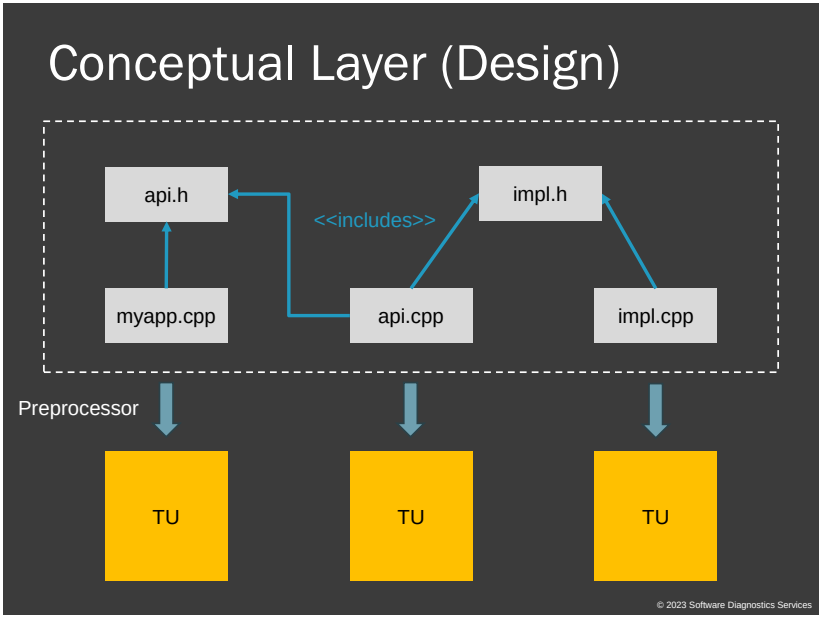


Named namespaces allow fine-grained scope isolation.

The code example:

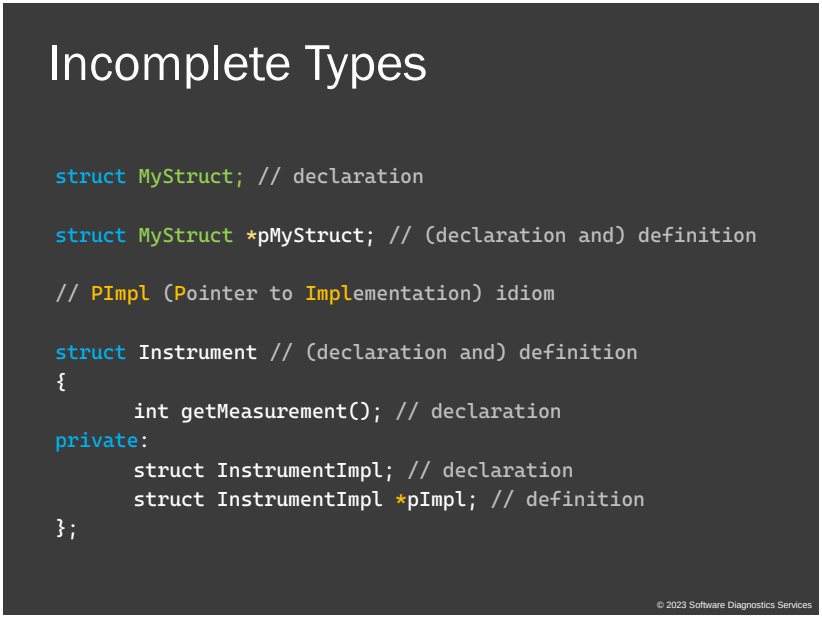
```
struct S { }; // via thirdparty.h
namespace mycode { struct S { }; }
//...
mycode::S s;
using namespace mycode;
S s2; // ambiguous
```

Conceptual Layer (Design)



In the design layer, we may want to separate implementation details.

Incomplete Types



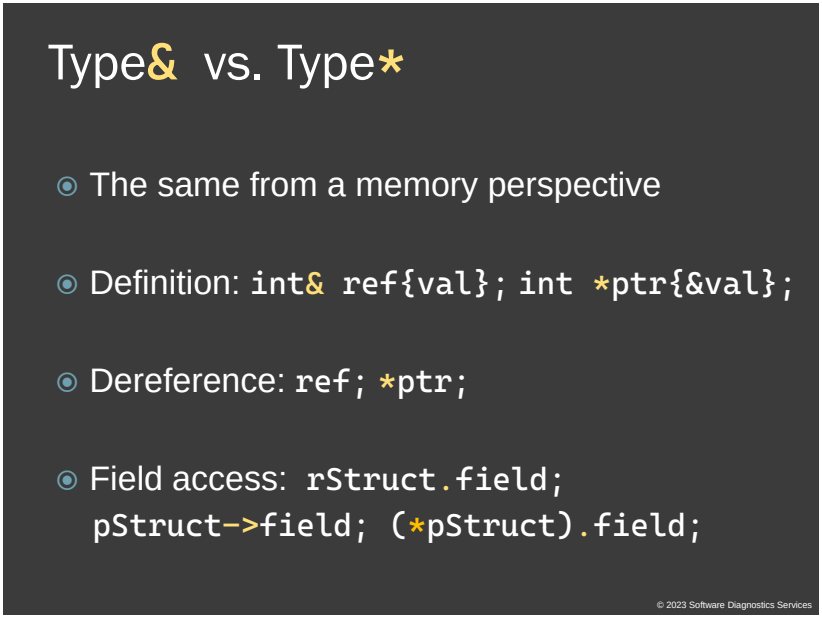
Such separation is achieved via incomplete types and the so-called **PImpl** (**P**ointer to **Impl**ementation) idiom:

```
struct MyStruct; // declaration

struct MyStruct *pMyStruct; // (declaration and) definition

// PImpl (Pointer to Implementation) idiom

struct Instrument // (declaration and) definition
{
        int getMeasurement(); // declaration
private:
        struct InstrumentImpl; // declaration
        struct InstrumentImpl *pImpl; // definition
};
```

References

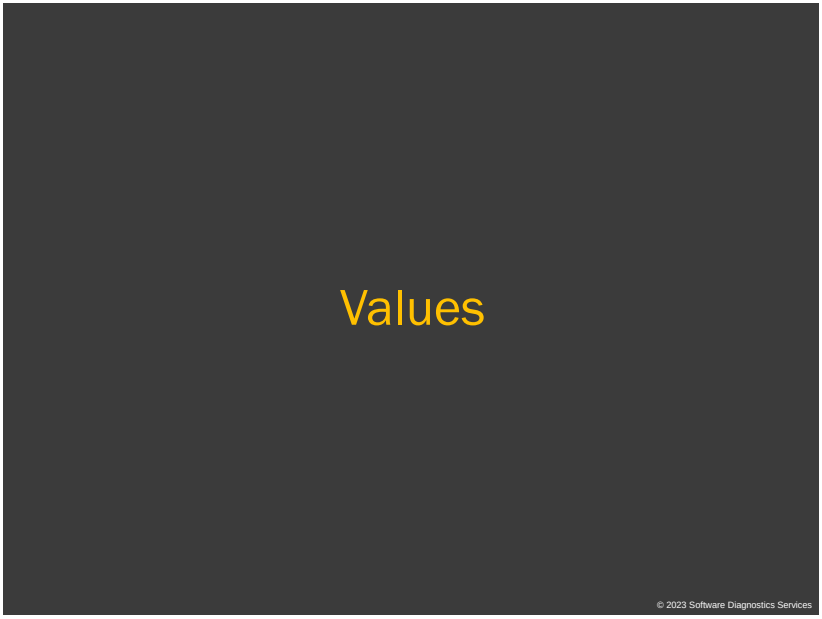
Now, a slide for C++ references. We plan to extend this section in the second edition.

Type& vs. Type*



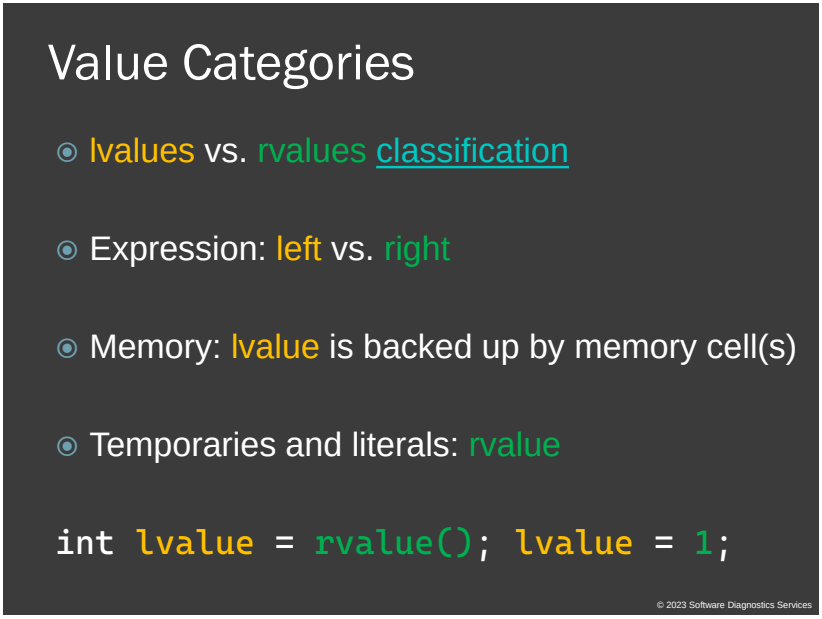
From the memory perspective, references and pointers are the same thing. The only difference is that you cannot have a dangling reference; it must be initialized.

Values



Let’s now briefly discuss various categories of values. These are what is stored in memory. A pointer value is also a value that is interpreted as a memory address pointing to some other value elsewhere.

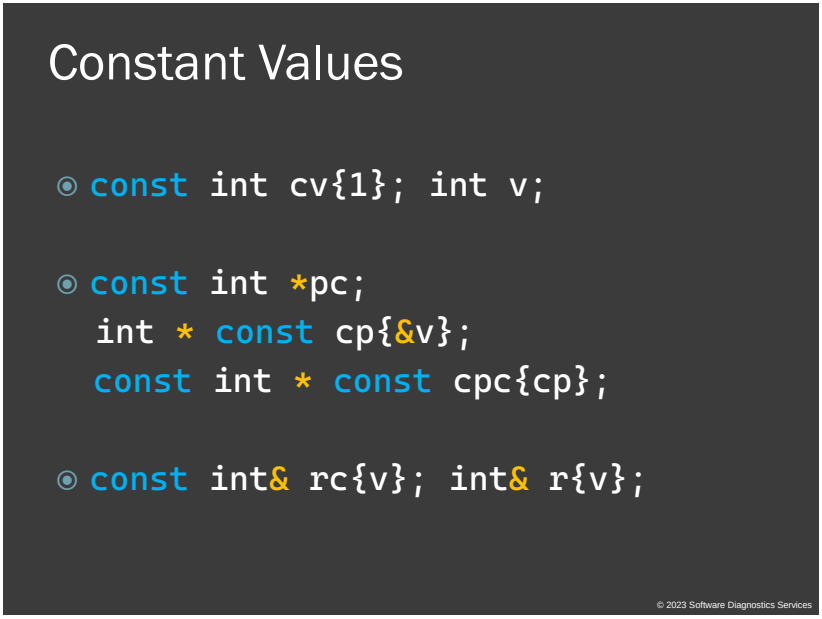
Value Categories



When reading serious C++ documentation, you frequently see the so-called **lvalues** and **rvalues** mentioned. Crudely, you can think about them as **l**eft and **r**ight values in expressions, where the **r**ight value can be temporary, and the **l**eft value has to be backed up by some memory.

Classification
https://en.cppreference.com/w/cpp/language/value_category

Constant Values

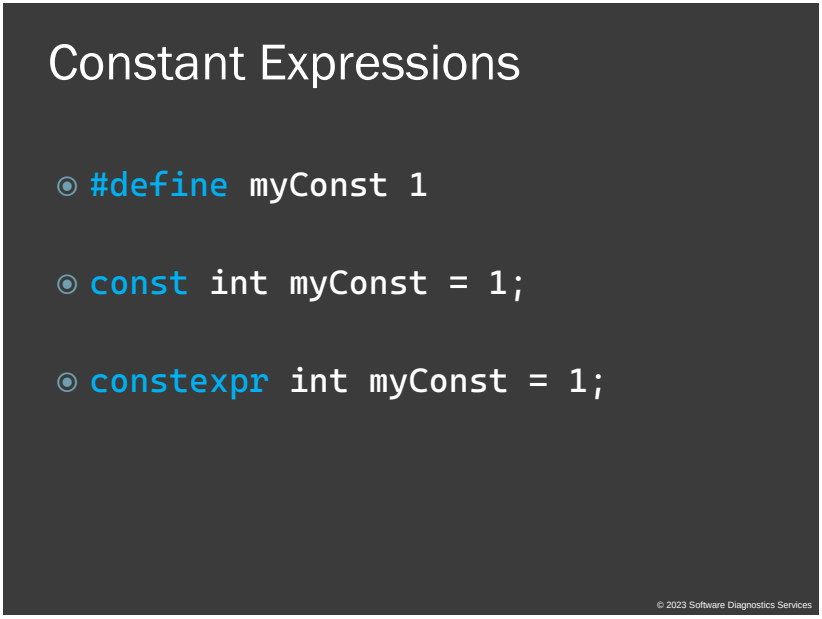


The values can also be constant, facilitating functional programming and code security. Please note that there can be pointers and references to constant values, constant pointers to mutable variables, and both. The way to read such declarations is from right to left.

The code examples:

```
const int cv{1}; int v;

const int *pc;
int * const cp{&v};
const int * const cpc{cp};

const int& rc{v}; int& r{v};
```

Constant Expressions



There are different ways to define constants for later symbolic use. The C and classic C++ way uses preprocessor (legacy) and `const`. The modern way is to use `constexpr`, which is more flexible.

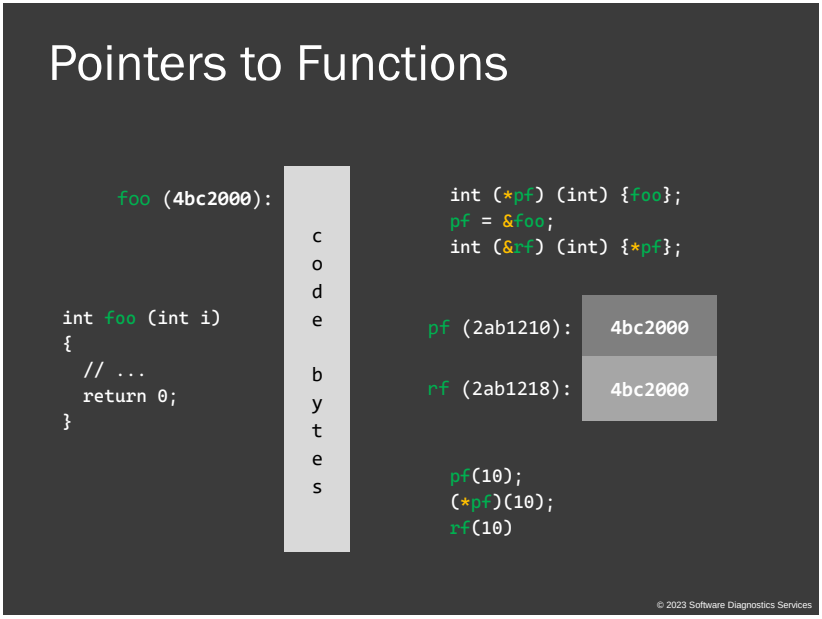
Code examples:

```
#define myConst 1

const int myConst = 1;

constexpr int myConst = 1;
```

Functions

This section is the largest in the course. We may split it up in the second edition once it grows more.

Pointers to Functions



Functions are code bytes and, therefore, occupy some memory locations with their start addresses.

The code examples corresponding to the memory diagram:

```
int foo (int i)
{
  // ...
  return 0;
}
```

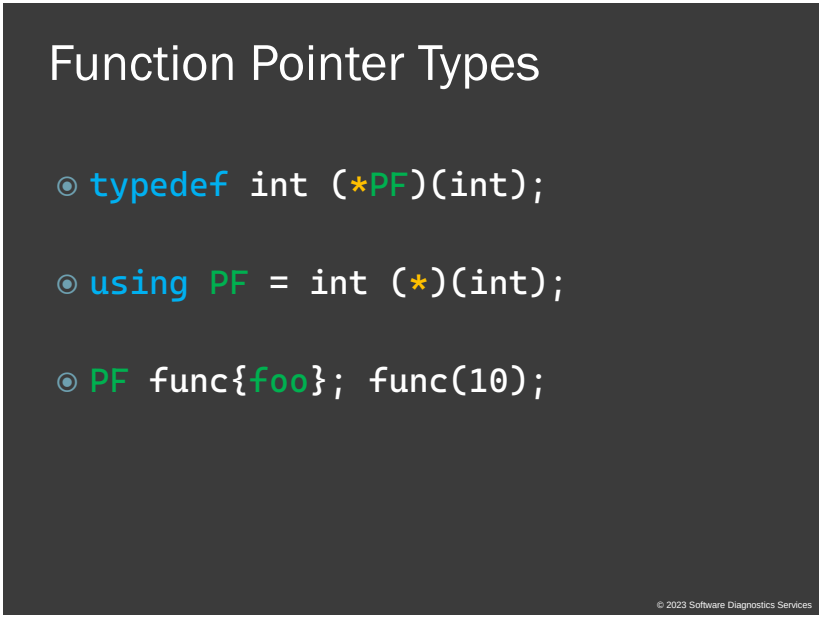
It is possible to have pointers and references to functions:

```
int (*pf) (int) {foo};
pf = &foo;
int (&rf) (int) {*pf};
```

When having a pointer or reference to a function, it is possible to call the function with or without using the dereferencing syntax (*):

```
pf(10);
(*pf)(10);
rf(10)
```

Function Pointer Types



Function pointer type declarations can be done using the classic `typedef` syntax or via the more modern `using` type alias.

The code examples:

```
typedef int (*PF)(int);

using PF = int (*)(int);

PF func{foo}; func(10);
```

Reading Declarations

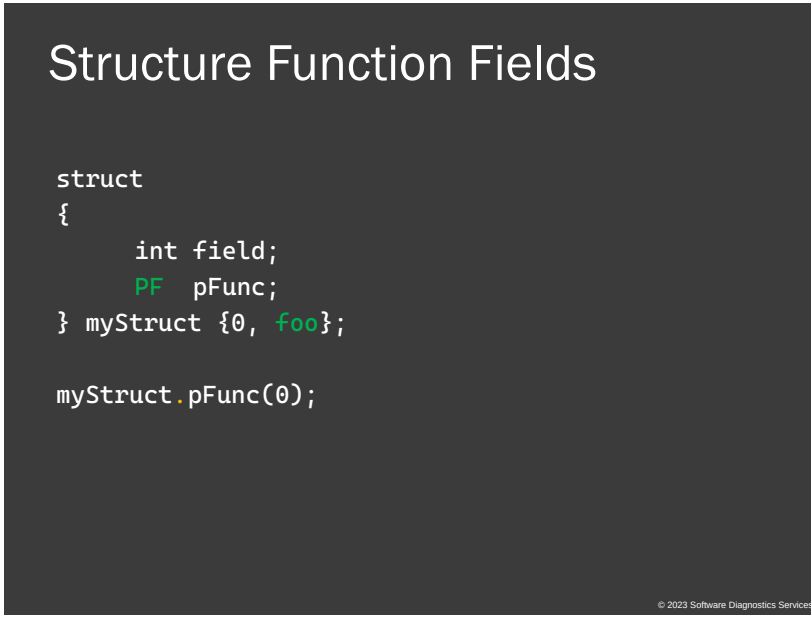
- Right → Left, $[]_{right}$ or $()_{right}$ → Right
- Examples:

```
const int* const* *arr[10];

int (*(*difficult)(int (*)(int), int))(int);
using DF = PF (*)(PF, int);
DF difficult2 {difficult};
```


It is worth knowing the rules of reading declarations since function pointer types can be quite complicated. The first example reads as an array of 10 elements, with each element a pointer to a pointer of constant pointers to constant integers. The next example is a pointer to a function that accepts a pointer to a function that accepts an integer and returns an integer, and accepts another integer, and returns a pointer to a function that accepts an integer and returns an integer. It can be simplified by using the common subtype PF. We can verify the compatibility of the two descriptions by initialization. GPT-4 is very good at deciphering such types.

```
const int* const* *arr[10];

int (*(*difficult)(int (*)(int), int))(int);
using DF = PF (*)(PF, int);
DF difficult2 {difficult};
```

Structure Function Fields

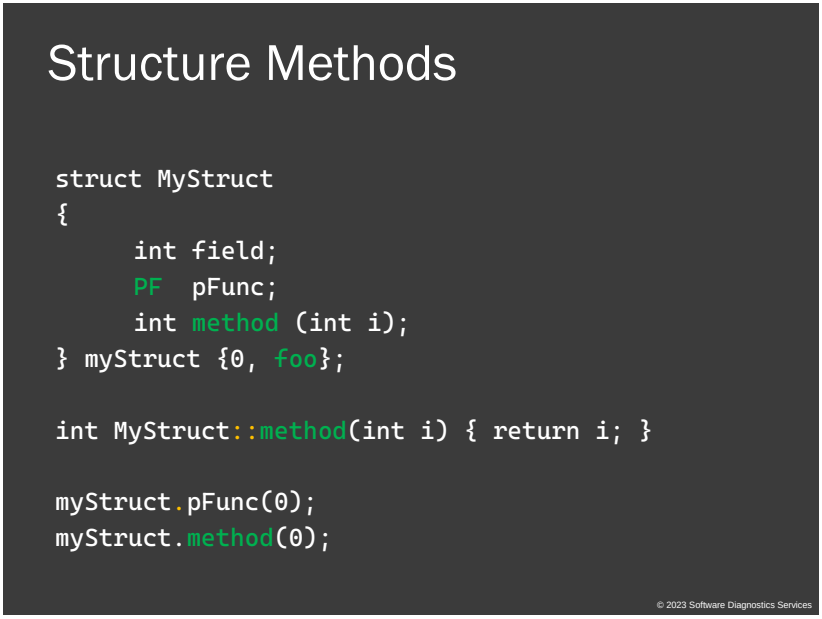


Structures may contain fields that are pointers to functions: the obvious way to implement OOP in C.

The code example:

```
struct
{
    int field;
    PF  pFunc;
} myStruct {0, foo};

myStruct.pFunc(0);
```

Structure Methods

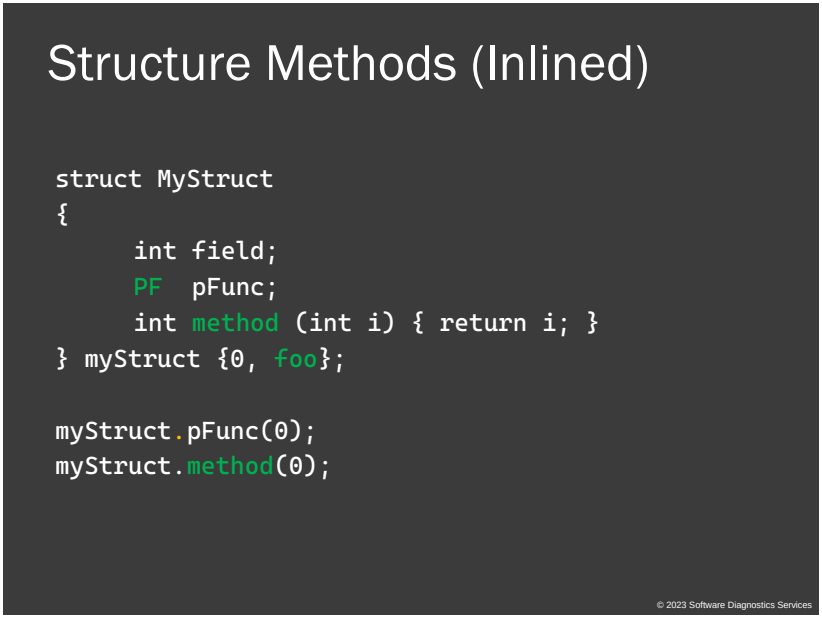


C++ introduced structure or class methods. They can be defined either inside the structure definition (next slide) or outside with the structure name qualification.

The code example:

```
struct MyStruct
{
    int field;
    PF  pFunc;
    int method (int i);
} myStruct {0, foo};

int MyStruct::method(int i) { return i; }

myStruct.pFunc(0);
myStruct.method(0);
```

Structure Methods (Inlined)

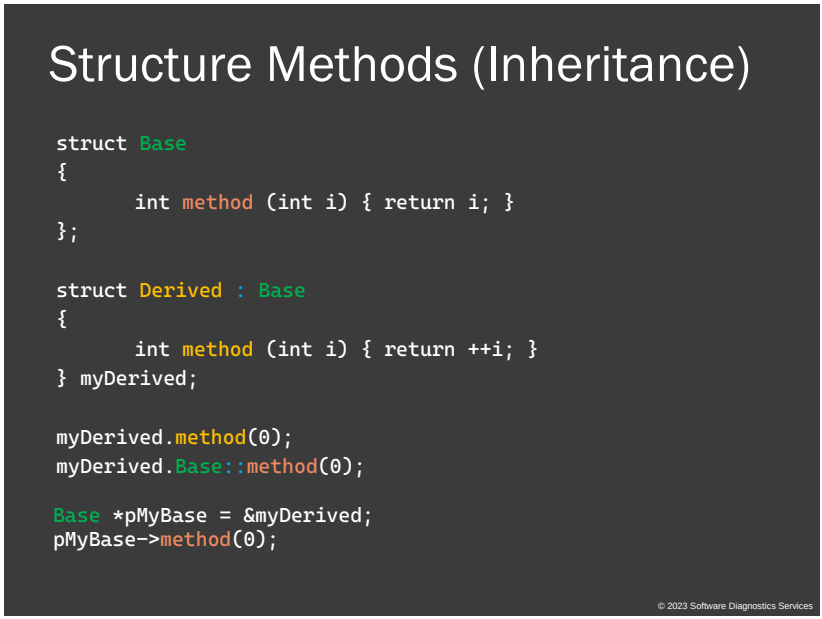


This is an alternative way to define short functions, although it exposes users to implementation details.

The code example:

```
struct MyStruct
{
    int field;
    PF  pFunc;
    int method (int i) { return i; }
} myStruct {0, foo};

myStruct.pFunc(0);
myStruct.method(0);
```

Structure Methods (Inheritance)

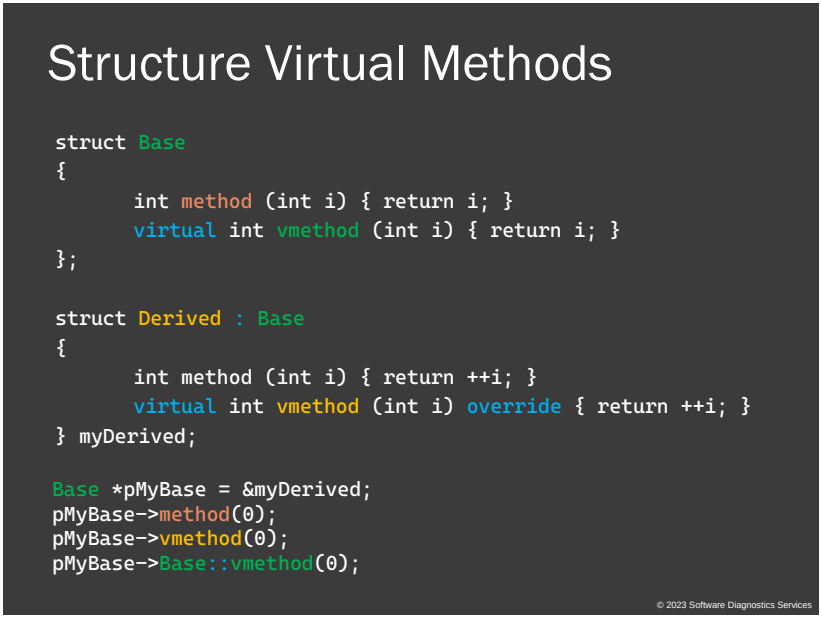


In the case of inheritance, like with fields, the derived structure methods hide methods with the same name in the base type unless explicit base type name qualification is used. However, when we have a pointer to a base type, then the base type method is called even if the actual object belongs to a derived type:

```
struct Base
{
    int method (int i) { return i; }
};

struct Derived : Base
{
    int method (int i) { return ++i; }
} myDerived;

myDerived.method(0);
myDerived.Base::method(0);
```

```
Base *pMyBase = &myDerived;
pMyBase->method(0);
```

Structure Virtual Methods

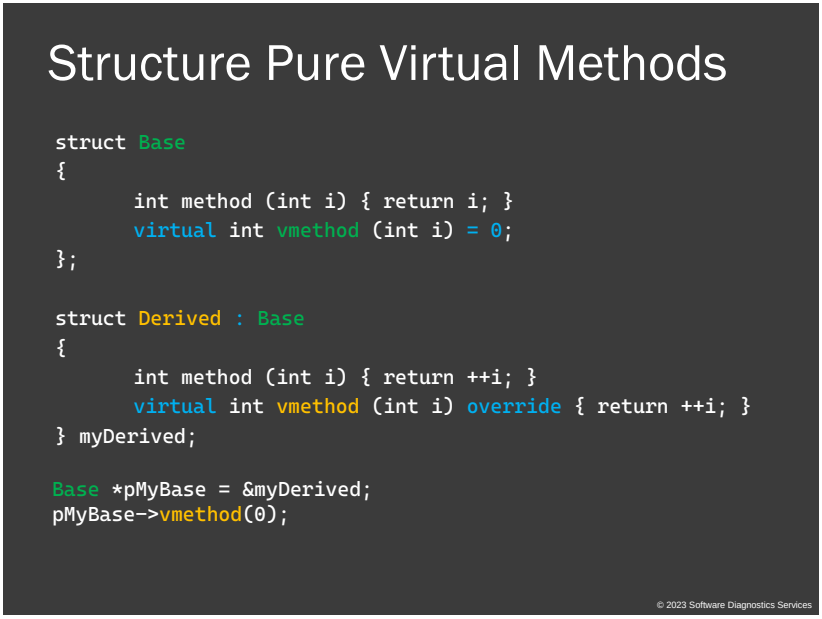


This previous slide problem is solved by introducing type-independent call virtual methods. In this case, the method of derived type is called when we have a pointer of a base type to it. The `override` specifier guarantees that we override the correct base method instead of introducing the new one by mistake:

```
struct Base
{
    int method (int i) { return i; }
    virtual int vmethod (int i) { return i; }
};

struct Derived : Base
{
    int method (int i) { return ++i; }
    virtual int vmethod (int i) override { return ++i; }
} myDerived;

Base *pMyBase = &myDerived;
pMyBase->method(0);
```

```
pMyBase->vmethod(0);
pMyBase->Base::vmethod(0);
```

Structure Pure Virtual Methods

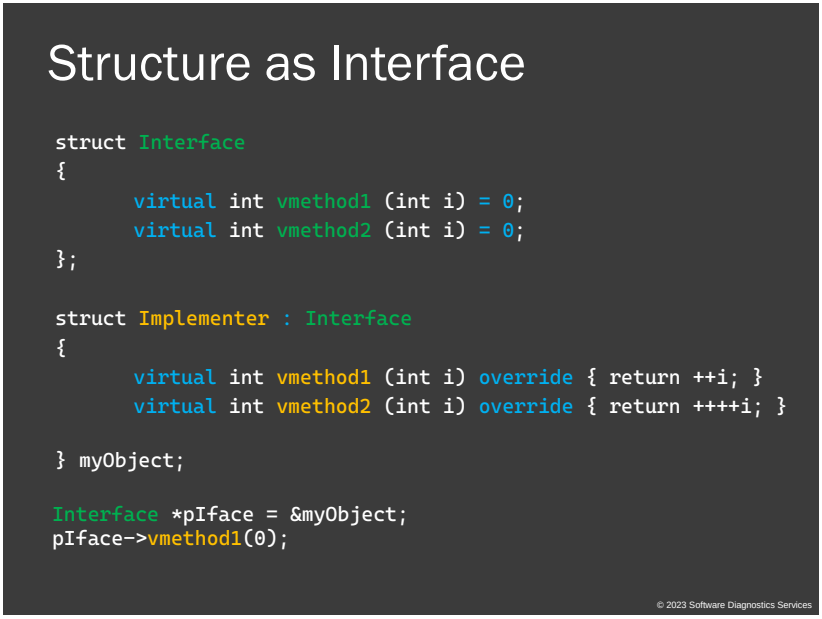


If we want to make sure we never define objects of the base type and make sure we override all virtual methods in the derived type, we can make the virtual functions **pure** by using `= 0`.

The code example:

```
struct Base
{
    int method (int i) { return i; }
    virtual int vmethod (int i) = 0;
};

struct Derived : Base
{
    int method (int i) { return ++i; }
    virtual int vmethod (int i) override { return ++i; }
} myDerived;

Base *pMyBase = &myDerived;
pMyBase->vmethod(0);
```

Structure as Interface

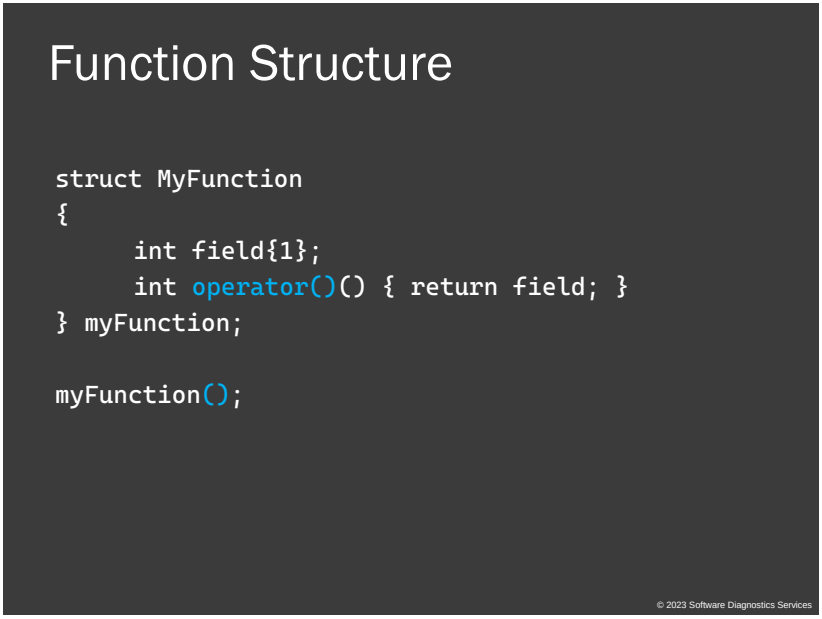


Pure virtual functions allow specifying abstract interfaces the derived types have to implement.

The code example:

```
struct Interface
{
    virtual int vmethod1 (int i) = 0;
    virtual int vmethod2 (int i) = 0;
};

struct Implementer : Interface
{
    virtual int vmethod1 (int i) override { return ++i; }
    virtual int vmethod2 (int i) override { return ++++i; }
} myObject;

Interface *pIface = &myObject;
pIface->vmethod1(0);
```

Function Structure

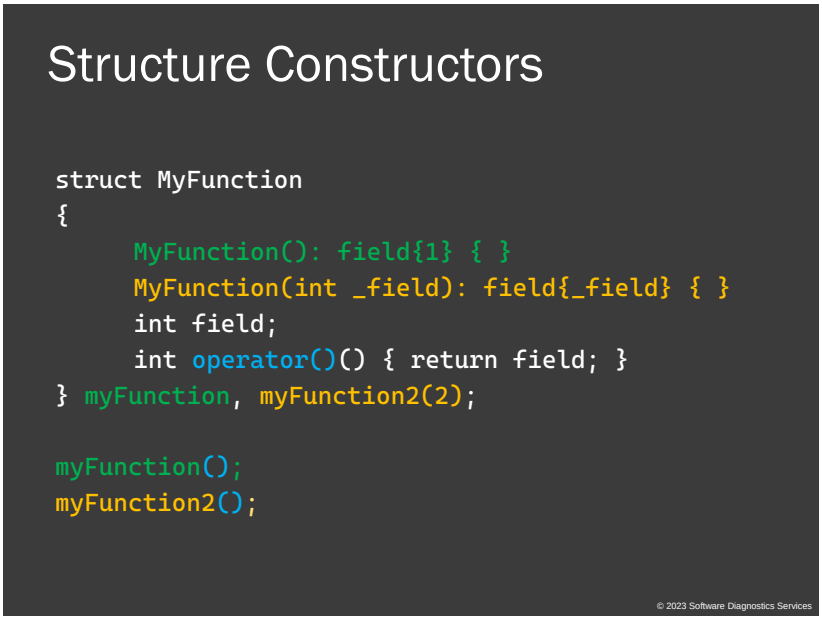


Functions may also encapsulate state. The best way to do it is via structures that implement function call operators.

The code example:

```
struct MyFunction
{
      int field{1};
      int operator()() { return field; }
} myFunction;

myFunction();
```

Structure Constructors

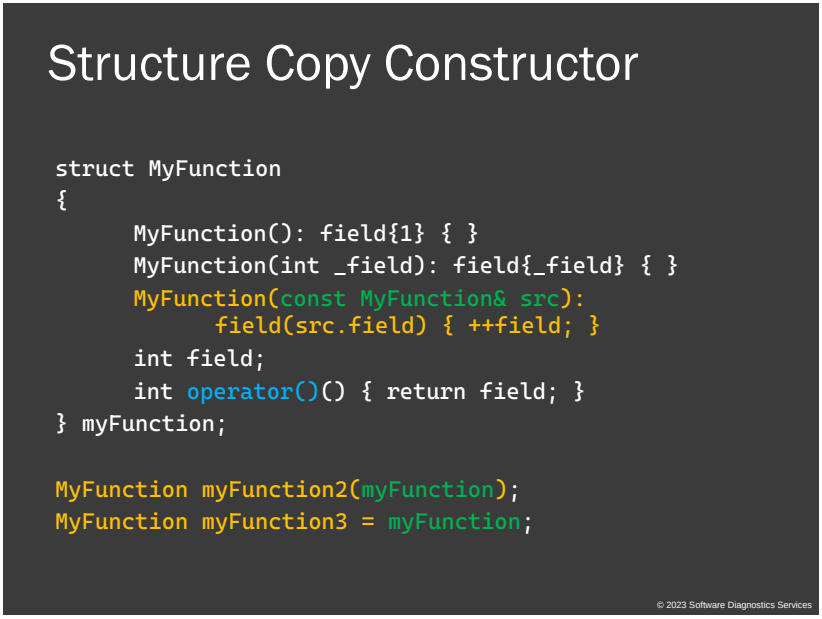


Now, we come to traditional OOP topics in classic C++. Constructors are methods with or without arguments for structure initialization with custom initialization logic inside, for example, acquiring required resources.

The code example:

```
struct MyFunction
{
    MyFunction(): field{1} { }
    MyFunction(int _field): field{_field} { }
    int field;
    int operator()() { return field; }
} myFunction, myFunction2(2);

myFunction();
myFunction2();
```

Structure Copy Constructor

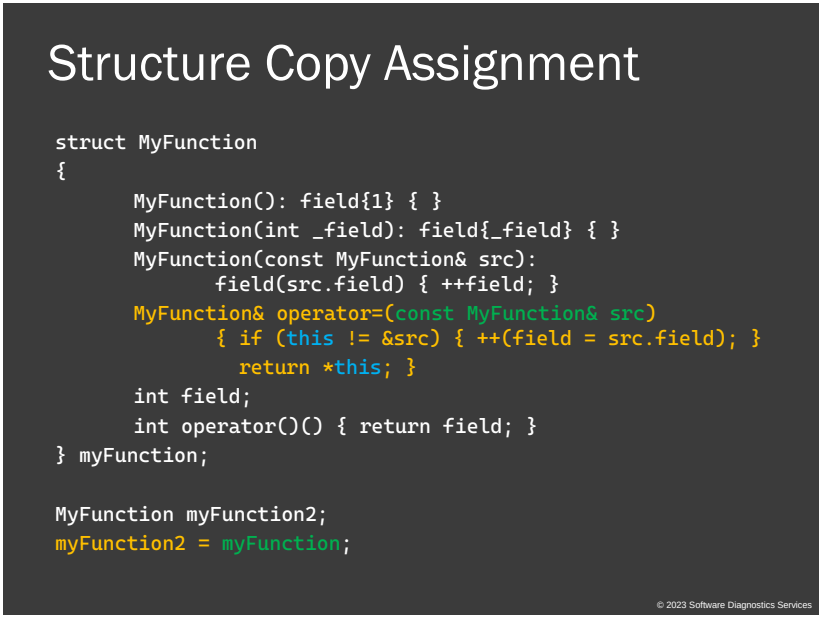


When we copy objects but need complex copying logic or nontrivial memory management copy constructor methods are quite handy. We pass the source object reference as const if we don't plan to modify it.

The code example:

```
struct MyFunction
{
    MyFunction(): field{1} { }
    MyFunction(int _field): field{_field} { }
    MyFunction(const MyFunction& src):
        field(src.field) { ++field; }
    int field;
    int operator()() { return field; }
} myFunction;

MyFunction myFunction2(myFunction);
MyFunction myFunction3 = myFunction;
```

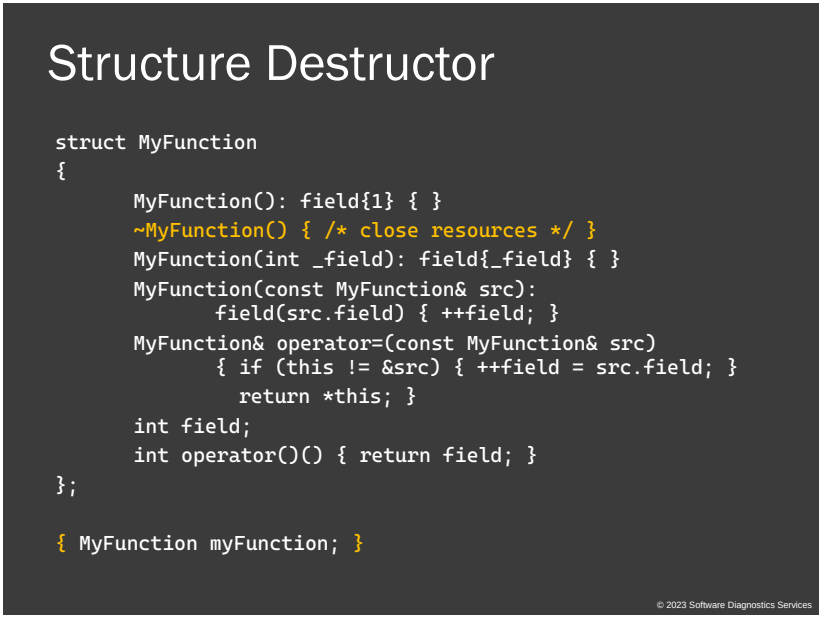
Structure Copy Assignment



In the case of nontrivial assignments, we can implement an assignment operator. We, however, should be careful not to copy to itself (`this`), and we return the non-const reference to itself (`*this`) to allow chained copies.

```
struct MyFunction
{
    MyFunction(): field{1} { }
    MyFunction(int _field): field{_field} { }
    MyFunction(const MyFunction& src):
        field(src.field) { ++field; }
    MyFunction& operator=(const MyFunction& src)
      { if (this != &src) { ++(field = src.field); }
        return *this; }
    int field;
    int operator()() { return field; }
} myFunction;

MyFunction myFunction2;
myFunction2 = myFunction;
```

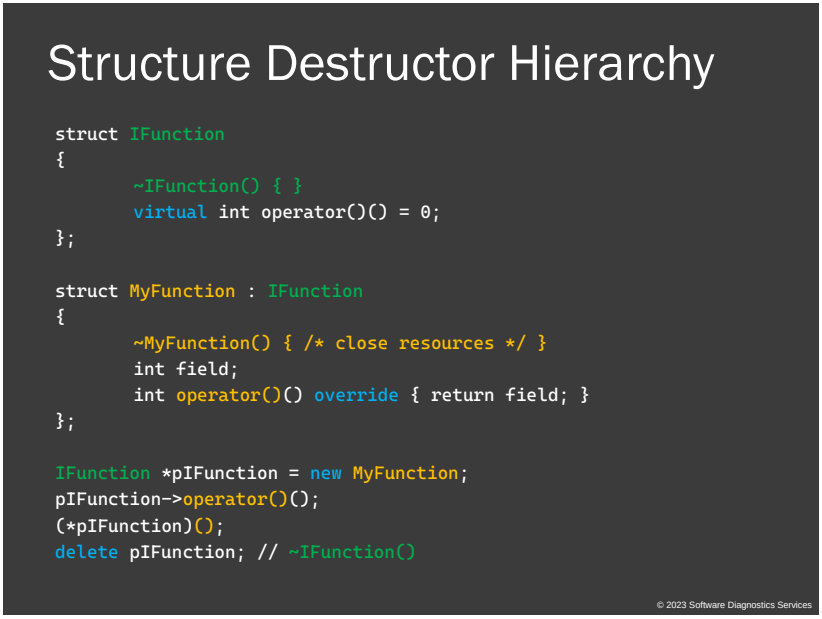
Structure Destructor



What if we want some complex logic, for example, releasing resources when the local object goes out of scope, or we delete it? Destructor is a method that is called automatically in such a case.

```
struct MyFunction
{
    MyFunction(): field{1} { }
    ~MyFunction() { /* close resources */ }
    MyFunction(int _field): field{_field} { }
    MyFunction(const MyFunction& src):
        field(src.field) { ++field; }
    MyFunction& operator=(const MyFunction& src)
      { if (this != &src) { ++field = src.field; }
        return *this; }
    int field;
    int operator()() { return field; }
};

{
    MyFunction myFunction;
}
```

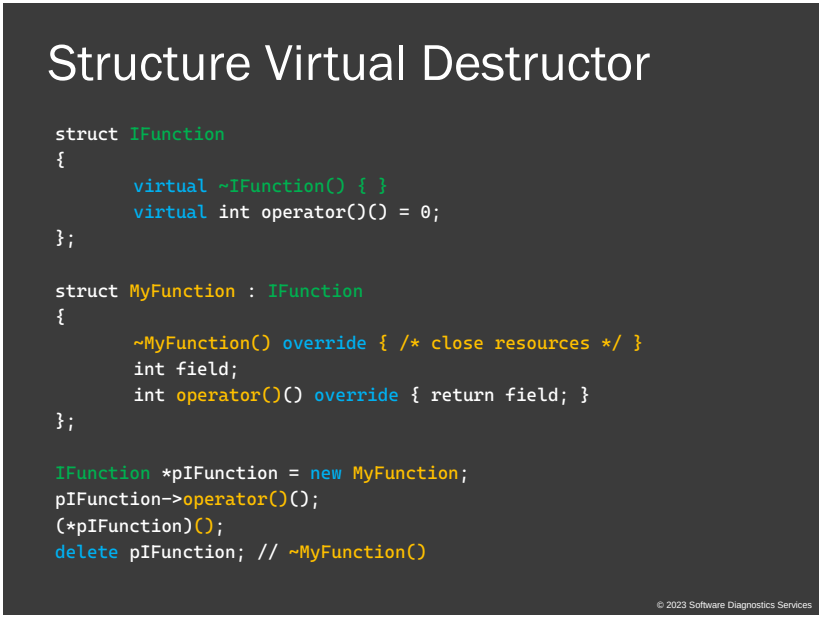
Structure Destructor Hierarchy



Conceptually, destructors are just like a normal method, so the wrong one may be called when we have a pointer of base type to an object of a derived type.

```
struct IFunction
{
    ~IFunction() { }
    virtual int operator()() = 0;
};

struct MyFunction : IFunction
{
    ~MyFunction() { /* close resources */ }
    int field;
    int operator()() override { return field; }
};

IFunction *pIFunction = new MyFunction;
pIFunction->operator()();
(*pIFunction)();
delete pIFunction; // ~IFunction()
```

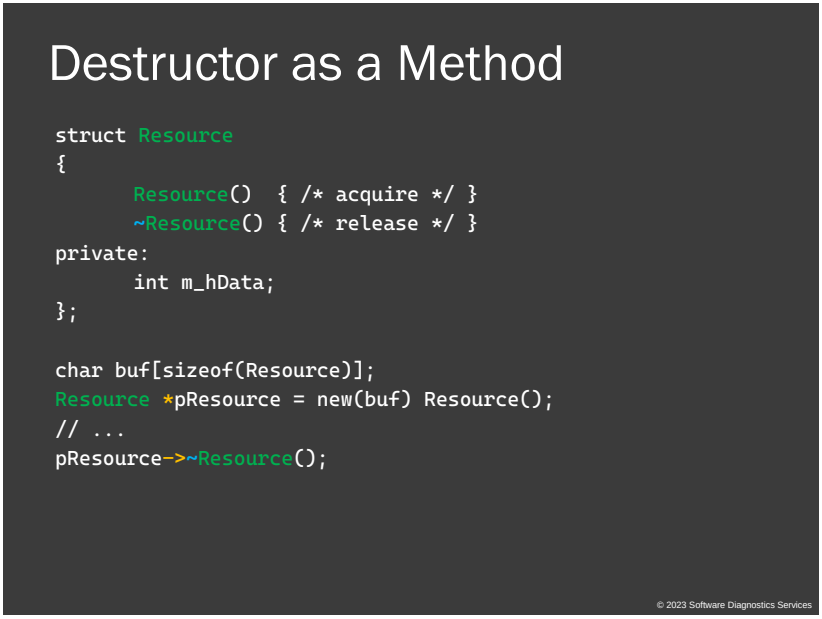
Structure Virtual Destructor



To make sure that the correct destructors are called, it is recommended to make them virtual too:

```
struct IFunction
{
    virtual ~IFunction() { }
    virtual int operator()() = 0;
};

struct MyFunction : IFunction
{
    ~MyFunction() override { /* close resources */ }
    int field;
    int operator()() override { return field; }
};

IFunction *pIFunction = new MyFunction;
pIFunction->operator()();
(*pIFunction)();
delete pIFunction; // ~MyFunction()
```

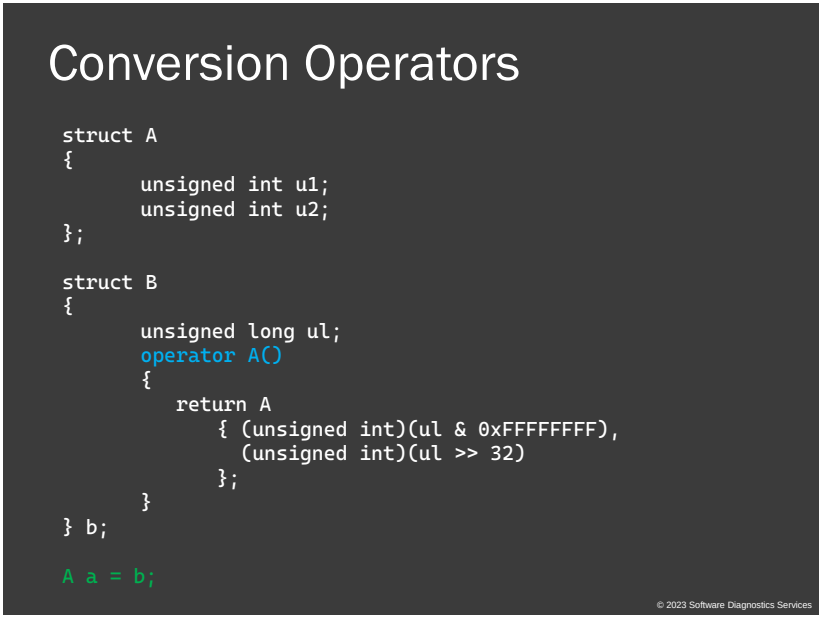
Destructor as a Method



Since a destructor is also a method, it is possible to call it directly in cases where we should not use standard delete methods, for example, when objects are allocated using placement `new`:

```
struct Resource
{
    Resource()  { /* acquire */ }
    ~Resource() { /* release */ }
private:
    int m_hData;
};

char buf[sizeof(Resource)];
Resource *pResource = new(buf) Resource();
// ...
pResource->~Resource();
```

Conversion Operators

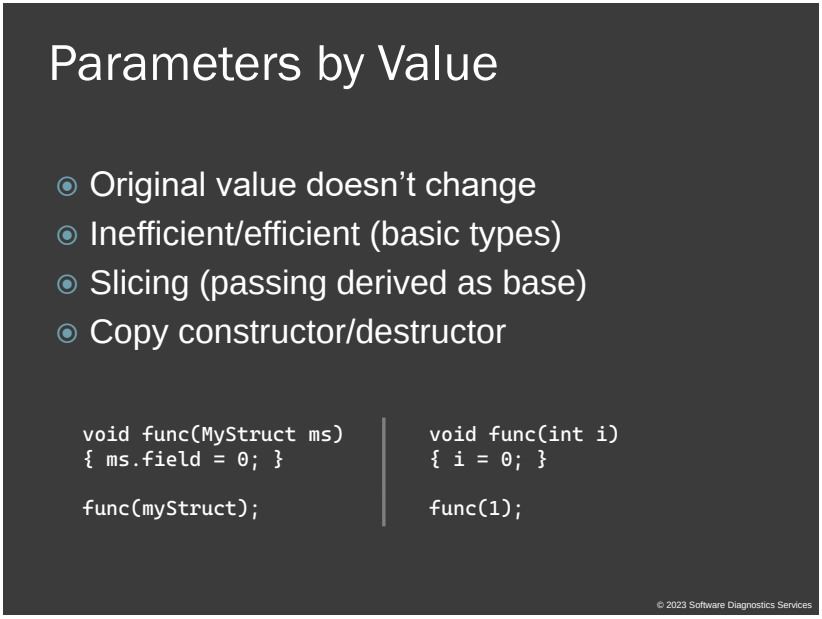


Sources for copy constructors and copy assignment operators are of the same type. What if we want to assign a different structure type? We can define custom conversion operators (it is also possible to use a “conversion” constructor):

```
struct A
{
    unsigned int u1;
    unsigned int u2;
};

struct B
{
    unsigned long ul;
    operator A()
    {
        return A
            { (unsigned int)(ul & 0xFFFFFFFF),
              (unsigned int)(ul >> 32)
            };
```

```
    }
} b;

A a = b;
```

Parameters by Value



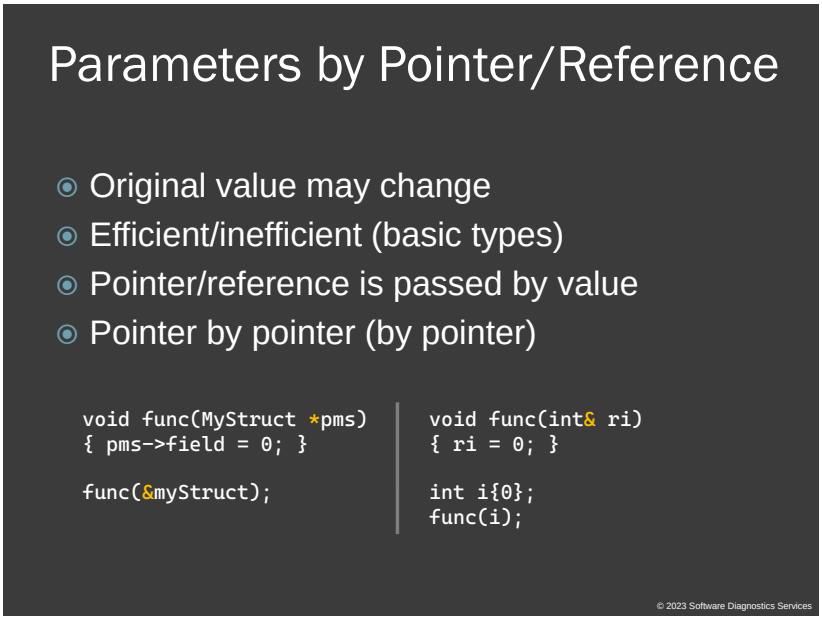
When we pass parameters by values, any modifications inside functions are lost once we return. Passing basic types by value is efficient, but passing structures are not unless they are very simple: various functions may be called, for example, copy constructors and destructors unless optimized by a compiler. There is also a possibility of slicing when inheritance is used.

The code examples:

```
void func(MyStruct ms)
{ ms.field = 0; }

func(myStruct);
```

```
void func(int i)
{ i = 0; }

func(1);
```

Parameters by Pointer/Reference



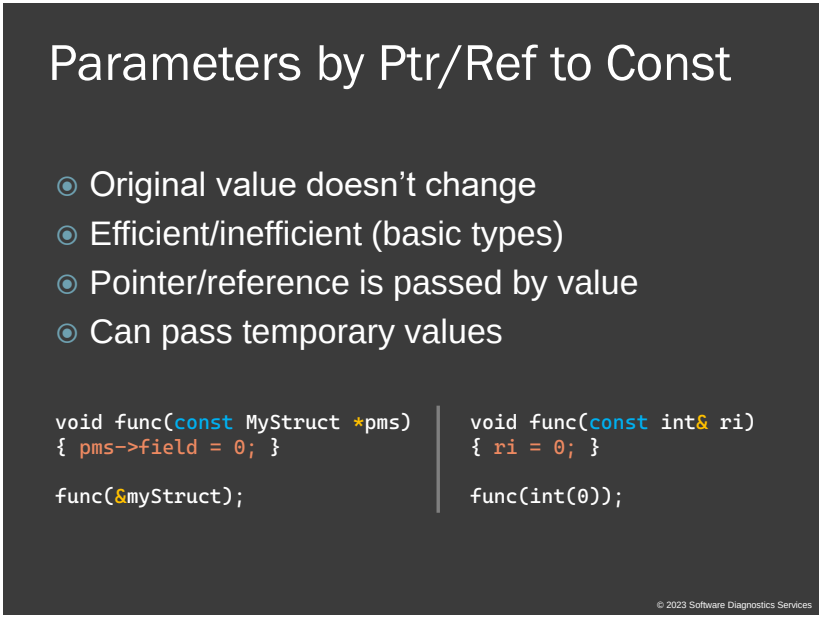
If we want efficiency for structures and also preserve changes to original values, we need to pass by reference. Again, this may be inefficient for basic types.

The code examples:

```
void func(MyStruct *pms)
{ pms->field = 0; }

func(&myStruct);
```

```
void func(int& ri)
{ ri = 0; }

int i{0};
func(i);
```

Parameters by Ptr/Ref to Const



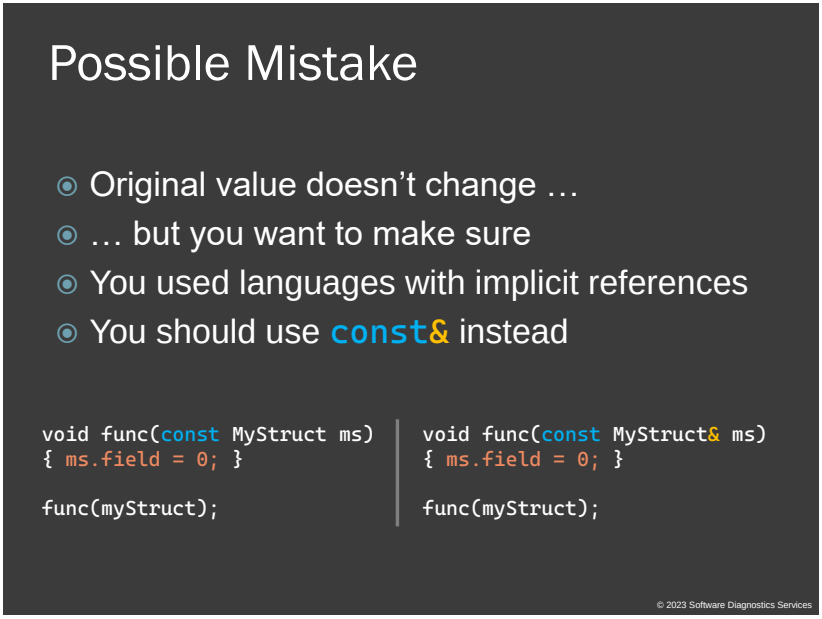
If we only want efficiency for structures, we need to pass by reference to `const`. In such a case, we also cannot modify the original values.

The code examples:

```
void func(const MyStruct *pms)
{ pms->field = 0; }

func(&myStruct);
```

```
void func(const int& ri)
{ ri = 0; }

func(int(0));
```

Possible Mistake

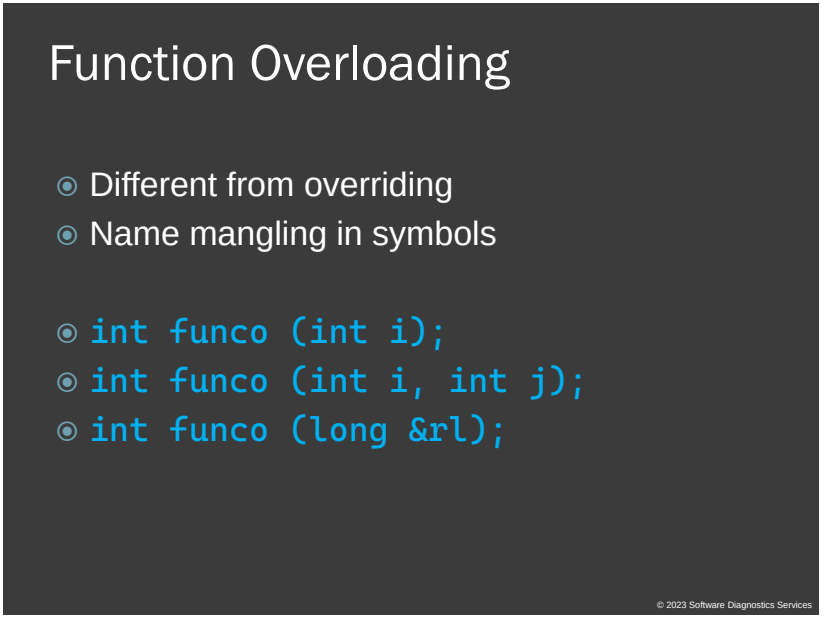


If you come from languages that use implicit references, you may omit `&` by mistake:

```
void func(const MyStruct ms)
{ ms.field = 0; }

func(myStruct);
```

```
void func(const MyStruct& ms)
{ ms.field = 0; }

func(myStruct);
```

Function Overloading

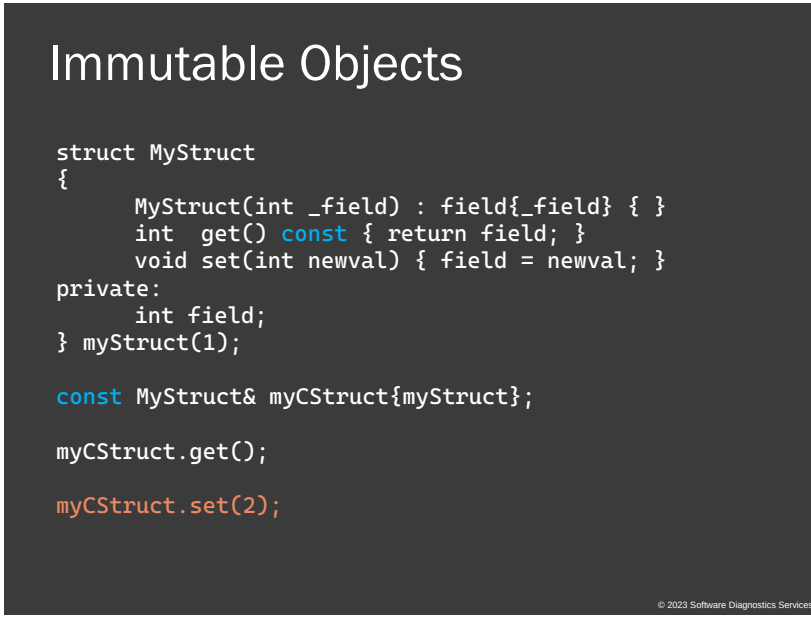


Function overloading allows reusing the same function names for functions with different numbers and types of parameters.

The code example:

```
int funco (int i);
int funco (int i, int j);
int funco (long &rl);
```

Immutable Objects

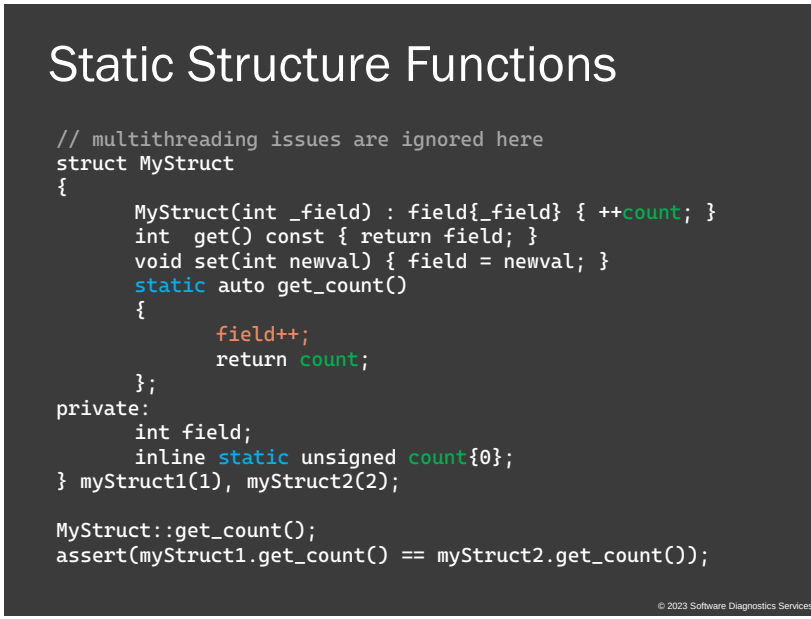


If you have `const` objects, you are only allowed to call methods that have the `const` specifier in their definition.

The code example:

```
struct MyStruct
{
    MyStruct(int _field) : field{_field} { }
    int  get() const { return field; }
    void set(int newval) { field = newval; }
private:
    int field;
} myStruct(1);

const MyStruct& myCStruct{myStruct};

myCStruct.get();

myCStruct.set(2);
```

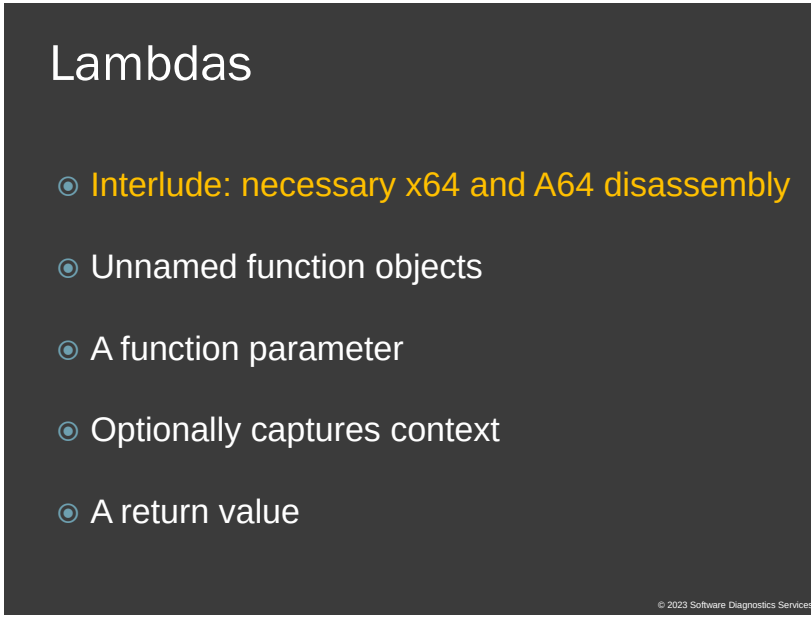
Static Structure Functions



Static structure functions are only allowed to access static structure fields shared among objects:

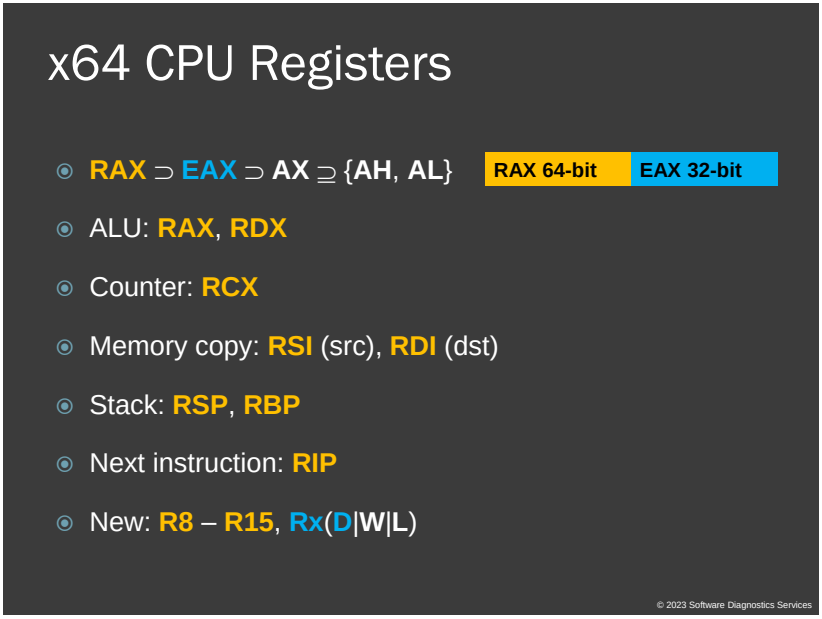
```
// multithreading issues are ignored here
struct MyStruct
{
    MyStruct(int _field) : field{_field} { ++count; }
    int  get() const { return field; }
    void set(int newval) { field = newval; }
    static auto get_count()
    {
        field++;
        return count;
    };
private:
    int field;
    inline static unsigned count{0};
} myStruct1(1), myStruct2(2);

MyStruct::get_count();
assert(myStruct1.get_count() == myStruct2.get_count());
```

Lambdas

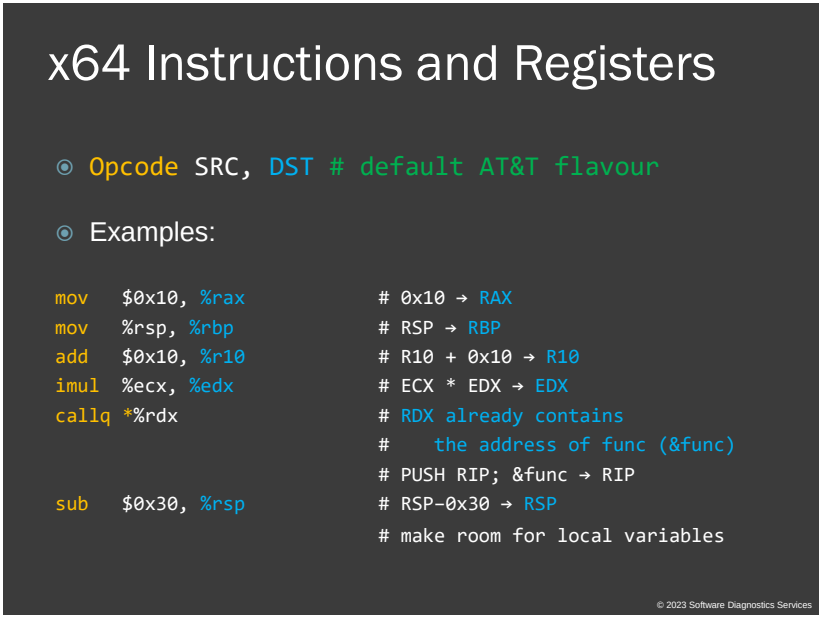


Before discussing lambdas introduced in C++11 and their internals, we take a brief tour of basic x64 and A64 disassembly.

x64 CPU Registers

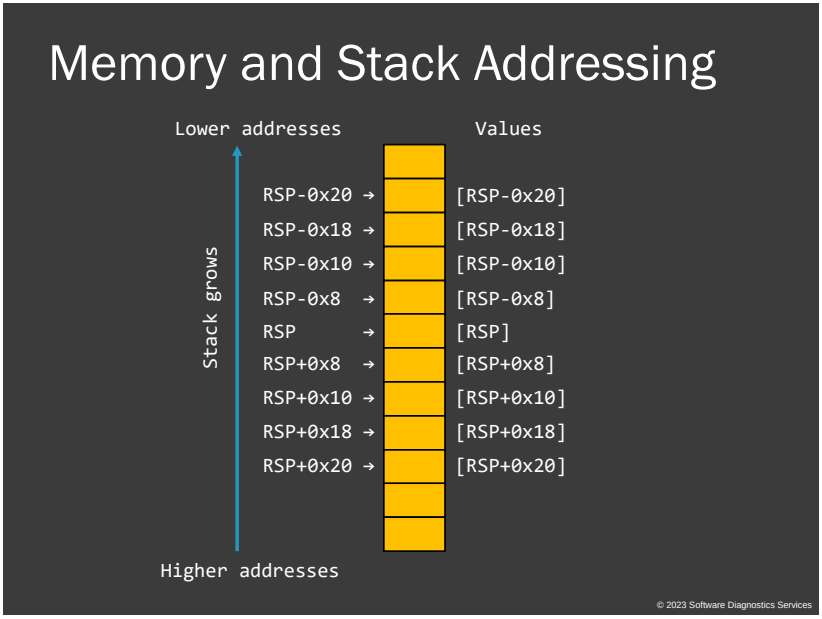


There are familiar 32-bit CPU register names, such as **EAX,** that are extended to 64-bit names, such as **RAX**. Most of them are traditionally specialized, such as ALU, counter, and memory copy registers. Although, now they all can be used as general-purpose registers. There is, of course, a stack pointer, **RSP**, and, additionally, a frame pointer, **RBP**, that is used to address local variables and saved parameters. It can be used for backtrace reconstruction. In some compiler code generation implementations, **RBP** is also used as a general-purpose register, with **RSP** taking the role of a frame pointer. An instruction pointer RIP is saved in the stack memory region with every function call, then restored on return from the called function. In addition, the x64 platform features another eight general-purpose registers, from **R8** to **R15**.



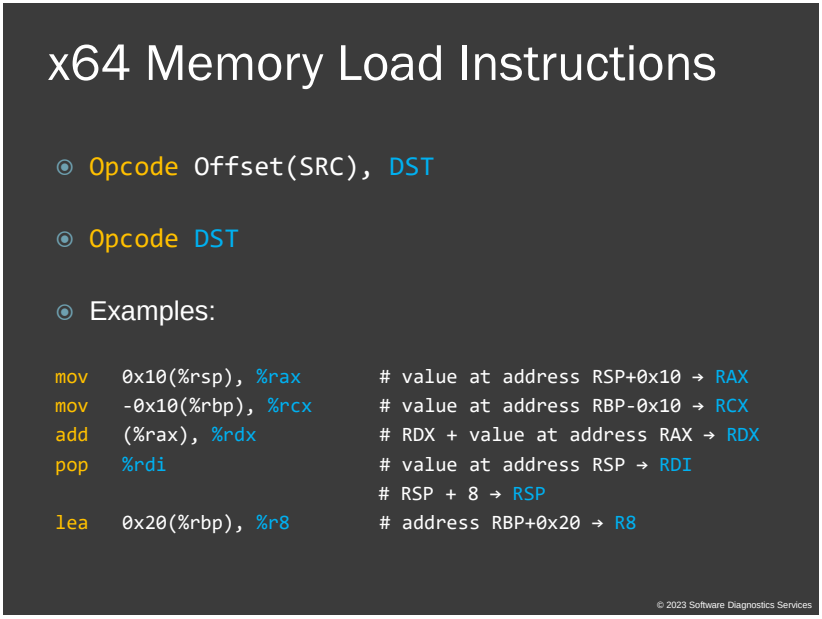
This slide shows a few examples of CPU instructions involving operations with registers, such as moving a value and doing arithmetic. The direction of operands is opposite to the Intel x64 disassembly flavor if you are accustomed to WinDbg on Windows. It is possible to use the Intel disassembly flavor in GDB, but we opted for the default AT&T flavor in line with our **Accelerated Linux Core Dump Analysis** and **Accelerated Linux Disassembly, Reconstruction, Reversing** books.

x64 Memory and Stack Addressing



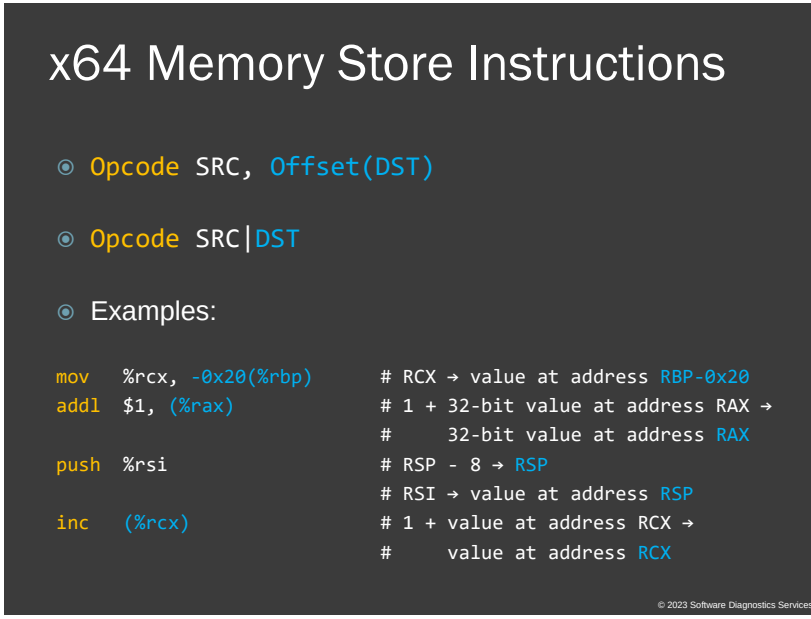
Before we look at operations with memory, let's look at a graphical representation of memory addressing. A thread stack is just any other memory region, so instead of **RSP** and **RBP,** any other register can be used. Please note that the stack grows towards lower addresses, so to access the previously pushed values, you need to use positive offsets from **RSP**.

x64 Memory Load Instructions



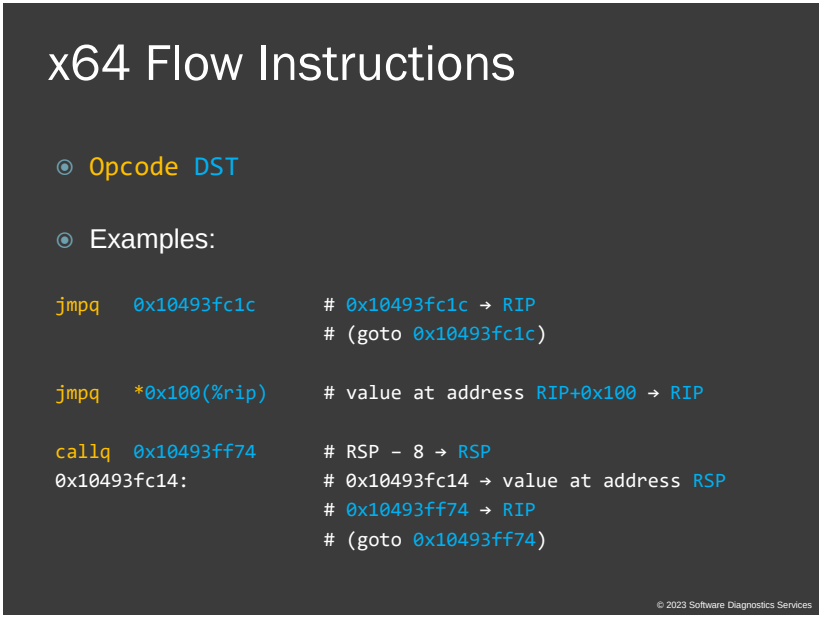
Constants are encoded in instructions, but if we need arbitrary values, we must get them from memory. Round brackets show memory access relative to an address stored in some register.

x64 Memory Store Instructions



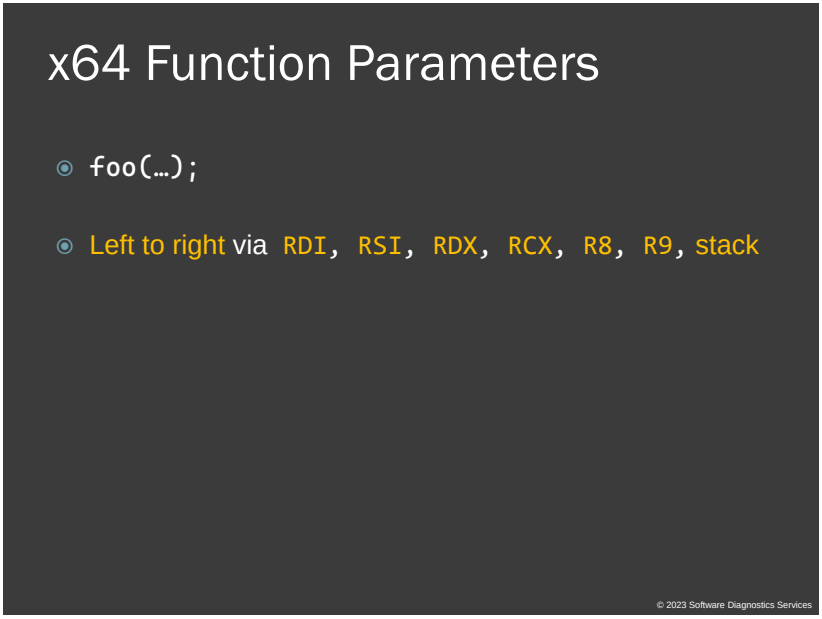
Storing is similar to loading.

x64 Flow Instructions



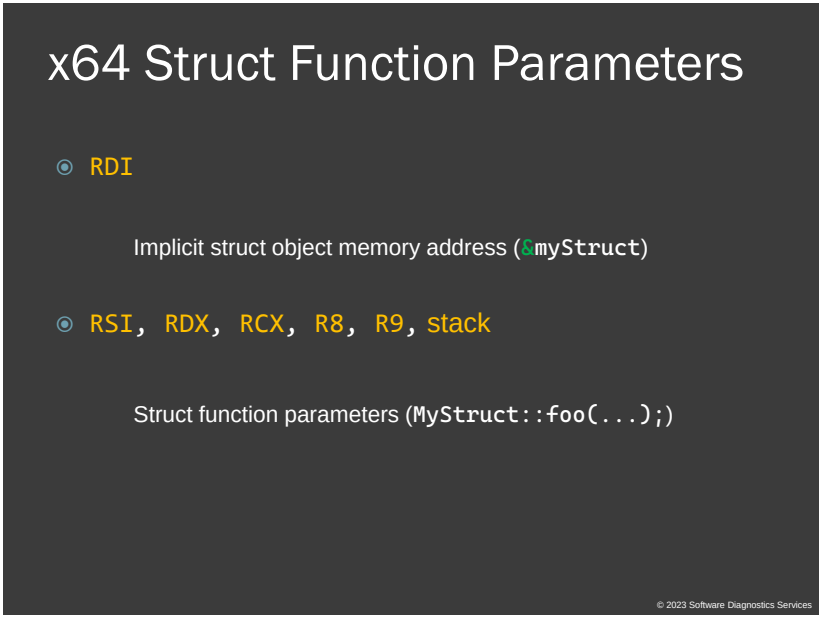
Goto (an unconditional jump) is implemented via the **JMP** instruction. Function calls are implemented via **CALL** instruction. For conditional branches, please look at the official Intel documentation.

x64 Function Parameters



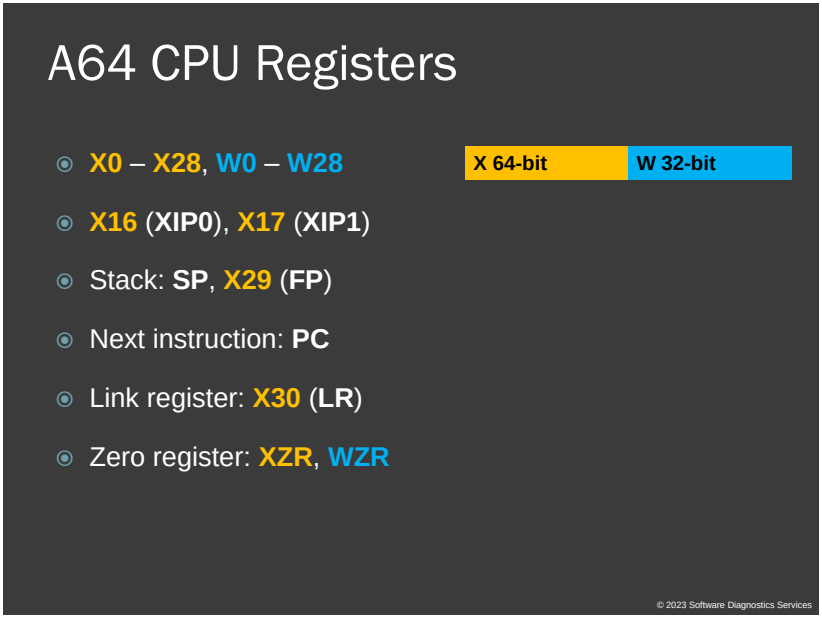
On the x64 Linux platform, the first six C and C++ function parameters from left to right are moved to CPU registers, and the rest are passed via stack locations.

x64 Struct Function Parameters

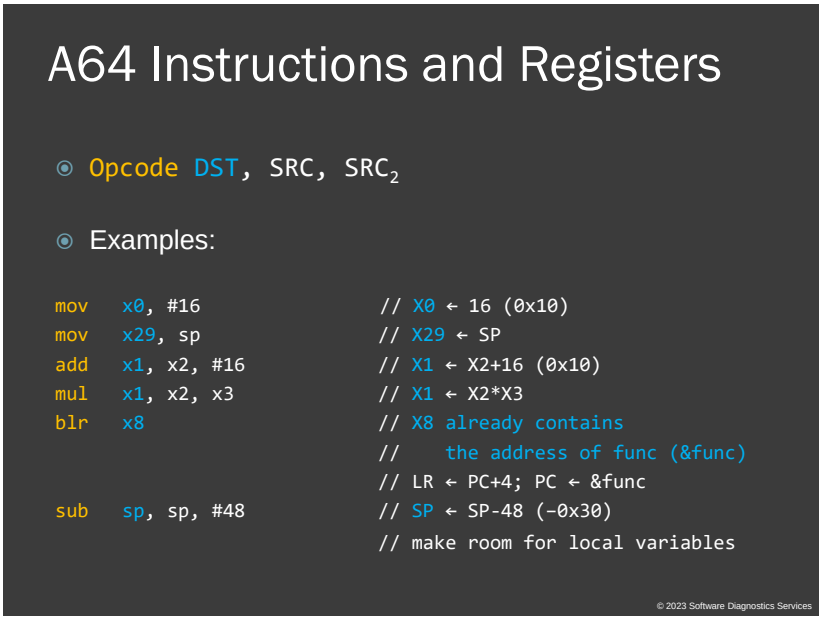


When an object struct nonstatic member function is called, the first parameter is implicit and, on the x64 Linux platform, is passed via **RDI**. It is an object address to help methods differentiate between objects of the same structure type and reference correct fields' memory. The rest of the parameters are passed as usual.

A64 CPU Registers

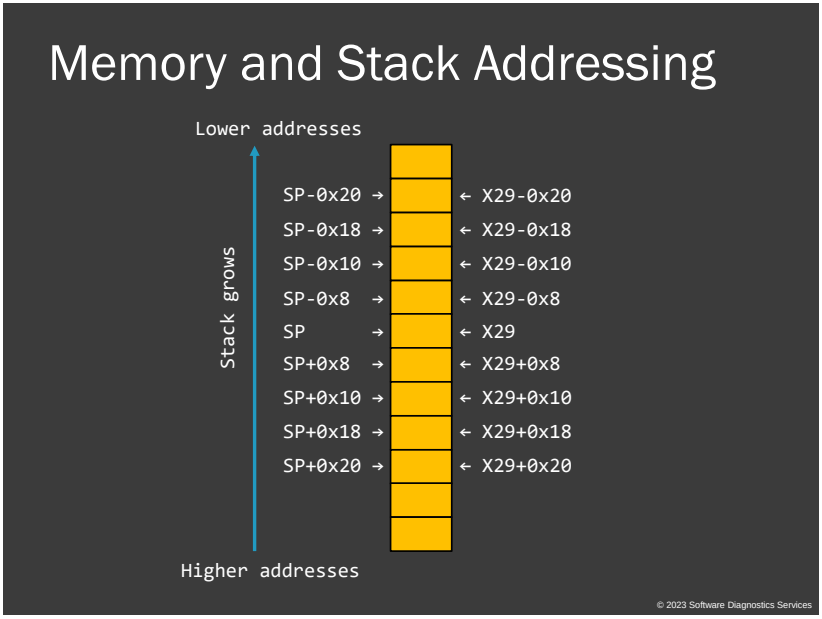


There are 31 general registers from **X0** and **X30**, with some delegated to specific tasks such as intra-procedure calls (**X16**, **XIP0**, and **X17**, **XIP1**), addressing stack frames (Frame Pointer, **FP**, **X29**) and return addresses, the so-called Link Register (**LR**, **X30**). When you call a function, the return address of a caller is saved in **LR**, not on the stack as in Intel/AMD x64. The return instruction in a callee uses the address in **LR** to assign it to **PC** and resume execution. But if a callee calls other functions, the current **LR** needs to be manually saved somewhere, usually on the stack. There's Stack Pointer, **SP**, of course. To get zero values, there's the so-called Zero Register, **XZR**. All **X** registers are 64-bit, and 32-bit lower parts are addressed via the **W** prefix. Next, we briefly look at some aspects related to our exercises.



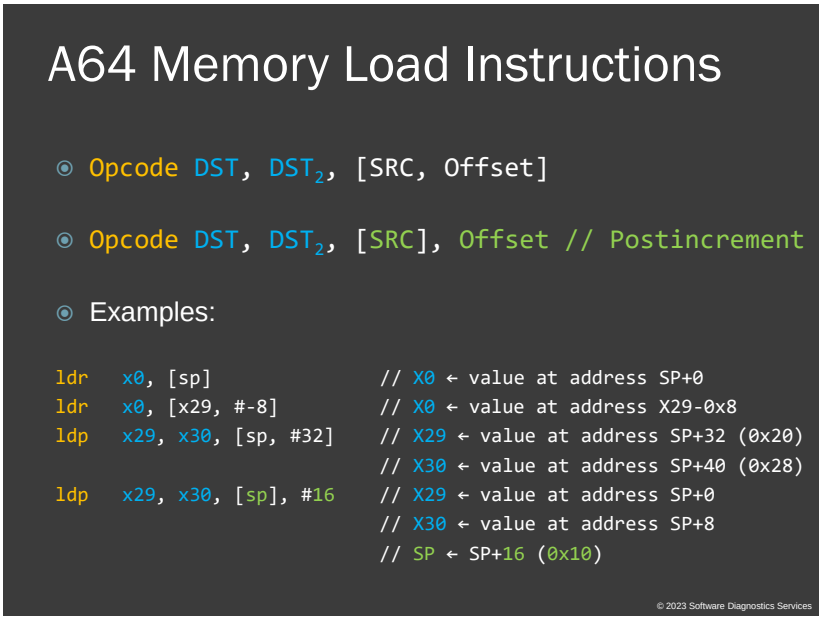
This slide shows a few examples of CPU instructions that involve operations with registers, for example, moving a value and doing arithmetic. The direction of operands is the same as in the Intel x64 disassembly flavor if you are accustomed to WinDbg on Windows. It is equivalent to an assignment. **BLR** is a call of some function whose address is in the register. **BL** means Branch and Link.

A64 Memory and Stack Addressing



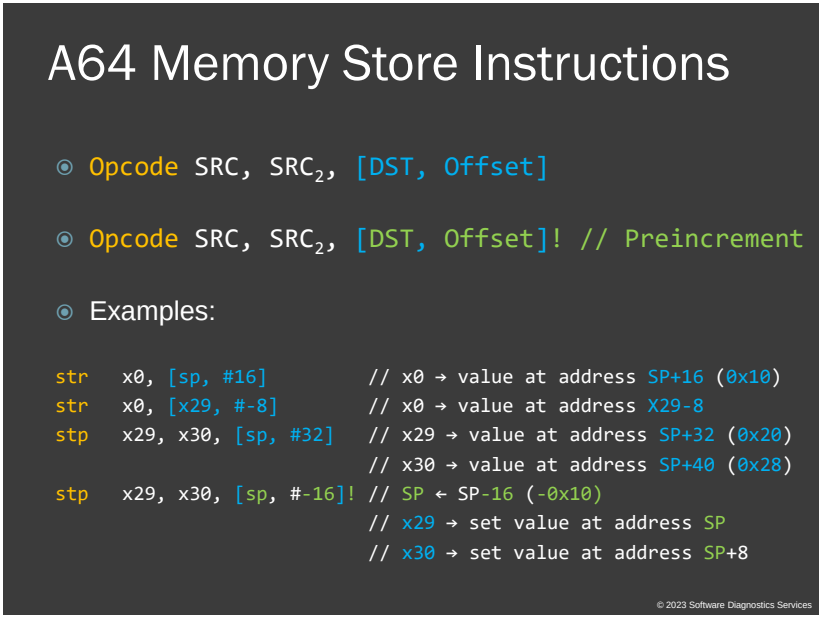
Before we look at operations with memory, let's look at a graphical representation of memory addressing. A thread stack is just any other memory region, so instead of **SP** and **X29** (**FP**), any other register can be used. Please note that the stack grows towards lower addresses, so to access the previously pushed values, you need to use positive offsets from **SP**.

A64 Memory Load Instructions



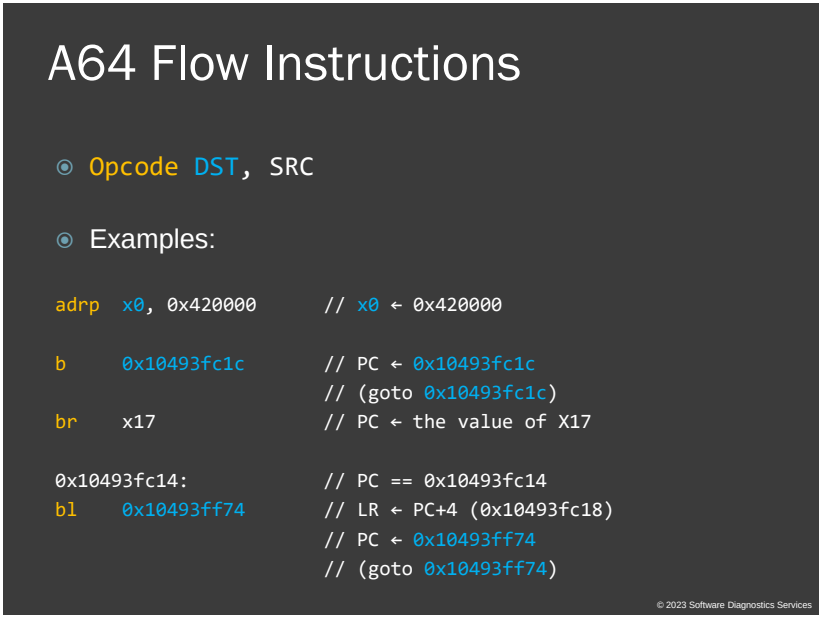
Constants are encoded in instructions, but if we need arbitrary values, we must get them from memory. Square brackets show memory access relative to an address stored in some register. There's also an option to adjust the value of the register after load, the so-called **Postincrement**, which can be negative.

A64 Memory Store Instructions



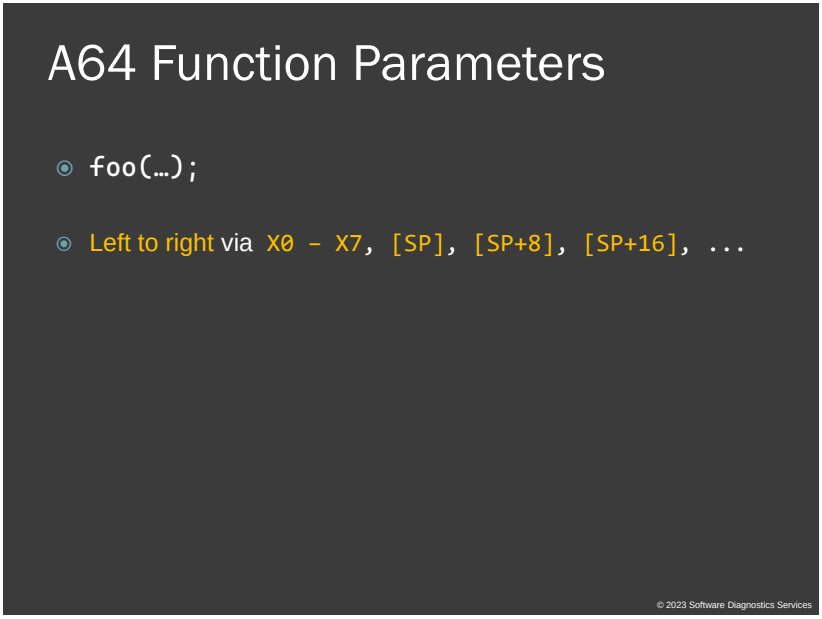
Storing operand order goes in the other direction compared to other instructions. There's a possibility to **Preincrement** the destination register before storing values.

A64 Flow Instructions



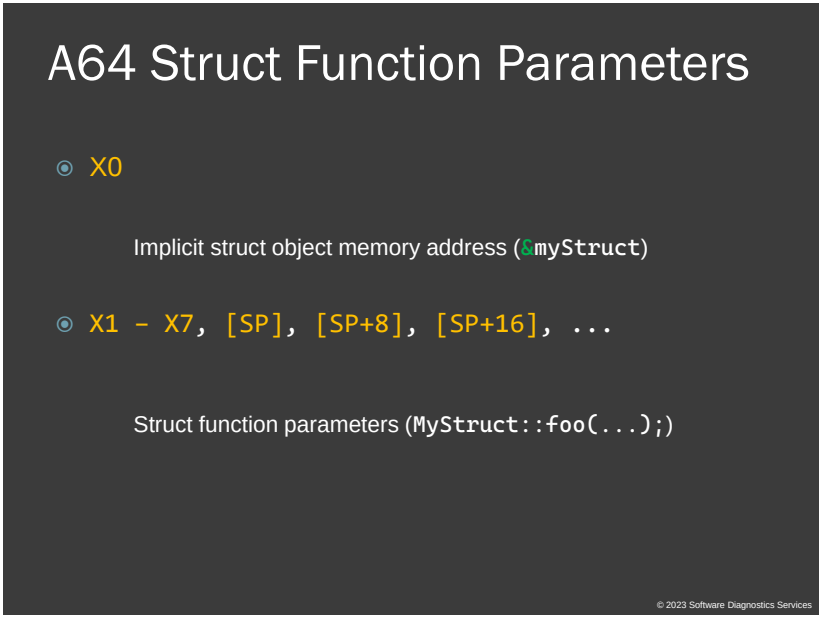
Because the size of every instruction is 4 bytes (32 bits), it is only possible to encode a part of a large 4GB address range, either as a relative offset to the current **PC** or via **ADRP** instruction. Goto (an unconditional branch) is implemented via the **B** instruction. Function calls are implemented via the **BL** (Branch and Link) instruction.

A64 Function Parameters



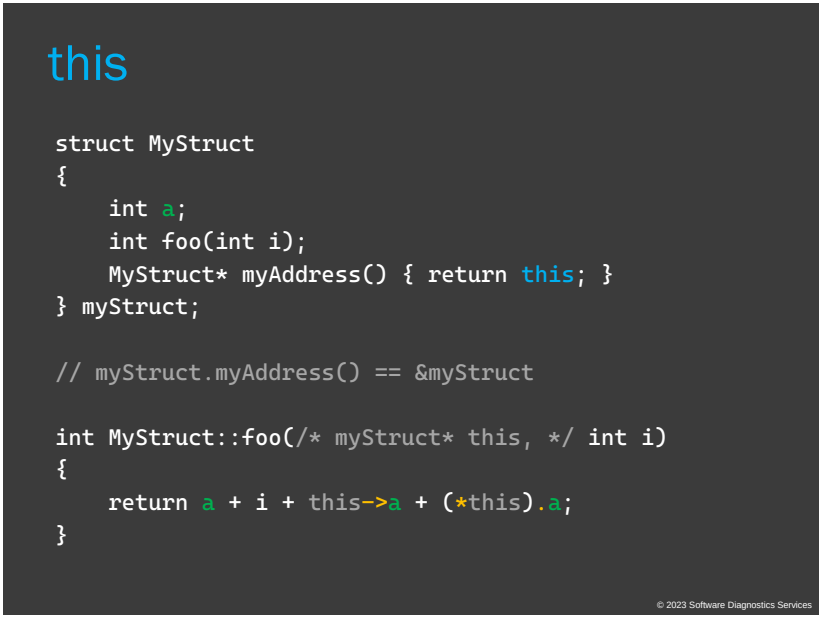
On the ARM64 Linux platform, the first eight parameters are passed via registers from left to right and the rest – via the stack locations.

A64 Struct Function Parameters



When an object struct nonstatic member function is called, the first parameter is implicit and, on the ARM64 Linux platform, is passed via **X0**. It is an object address to help methods differentiate between objects of the same structure type and reference correct fields' memory. The rest of the parameters are passed as usual.

this

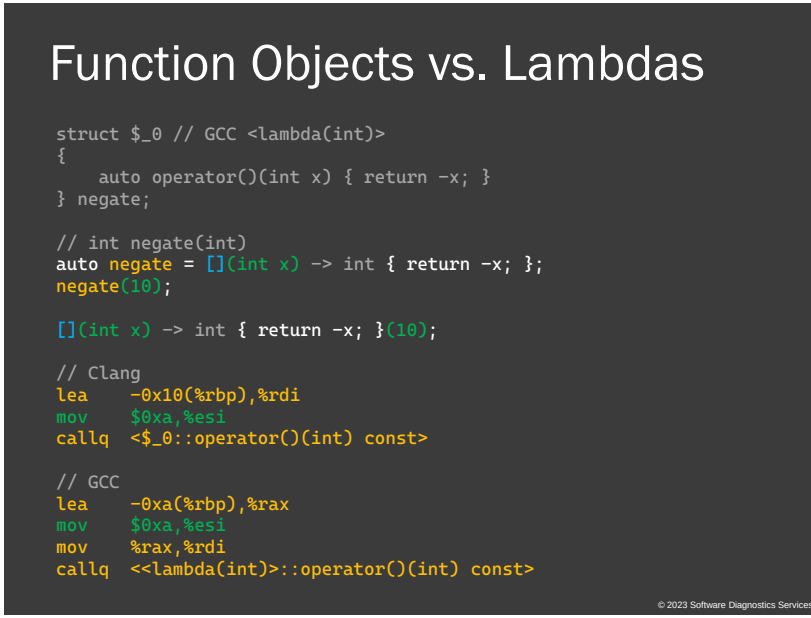


The address of the current object is contained in `this` pointer inside C++ source code. It can be used to refer to the current object fields and methods and can also be dereferenced.

The code example:

```
struct MyStruct
{
    int a;
    int foo(int i);
    MyStruct* myAddress() { return this; }
} myStruct;

// myStruct.myAddress() == &myStruct

int MyStruct::foo(/* myStruct* this, */ int i)
{
    return a + i + this->a + (*this).a;
}
```

Function Objects vs. Lambdas



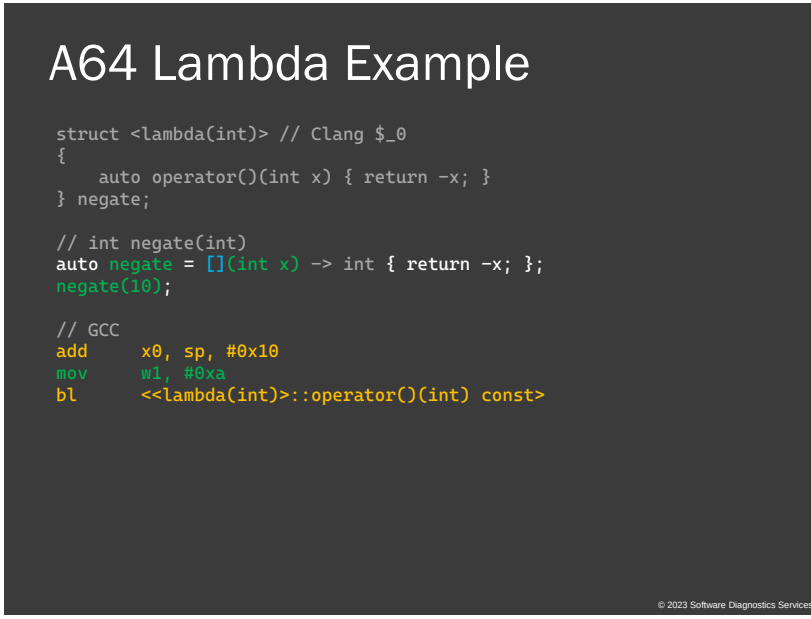
Lambdas are internally implemented as function objects.

The code example:

```
struct $_0 // GCC <lambda(int)>
{
    auto operator()(int x) { return -x; }
} negate;

// int negate(int)
auto negate = [](int x) -> int { return -x; };
negate(10);

[](int x) -> int { return -x; }(10);

// Clang
lea    -0x10(%rbp),%rdi
mov    $0xa,%esi
callq  <$_0::operator()(int) const>
```

```
// GCC
lea     -0xa(%rbp),%rax
mov     $0xa,%esi
mov     %rax,%rdi
callq   <<lambda(int)>::operator()(int) const>
```

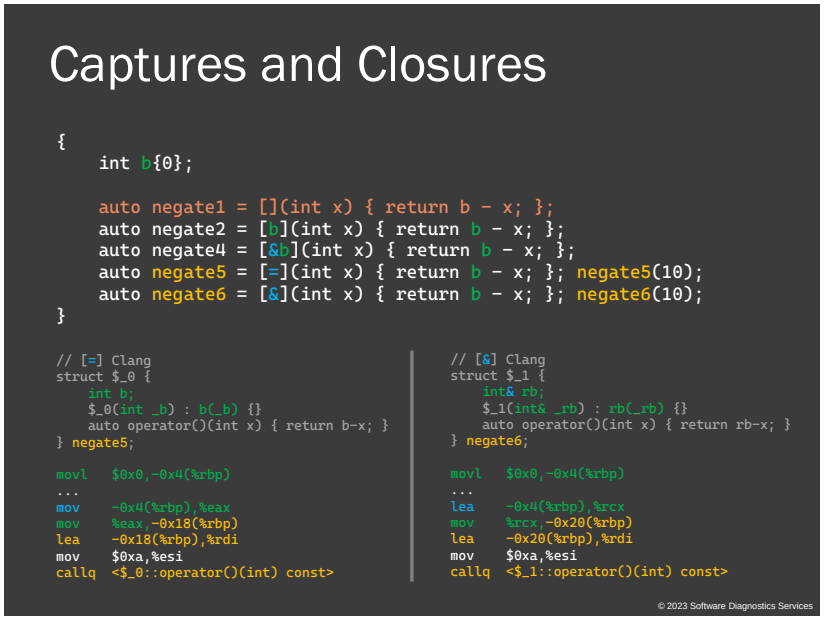
A64 Lambda Example



The ARM64 GCC disassembly fragment for the previous lambda example:

```
// GCC
add     x0, sp, #0x10
mov     w1, #0xa
bl      <<lambda(int)>::operator()(int) const>
```

Captures and Closures



Inside lambda code, it is possible to use local objects from the outer scope either by copy or by reference. This mechanism is internally implemented by lambda function objects.

The code example:

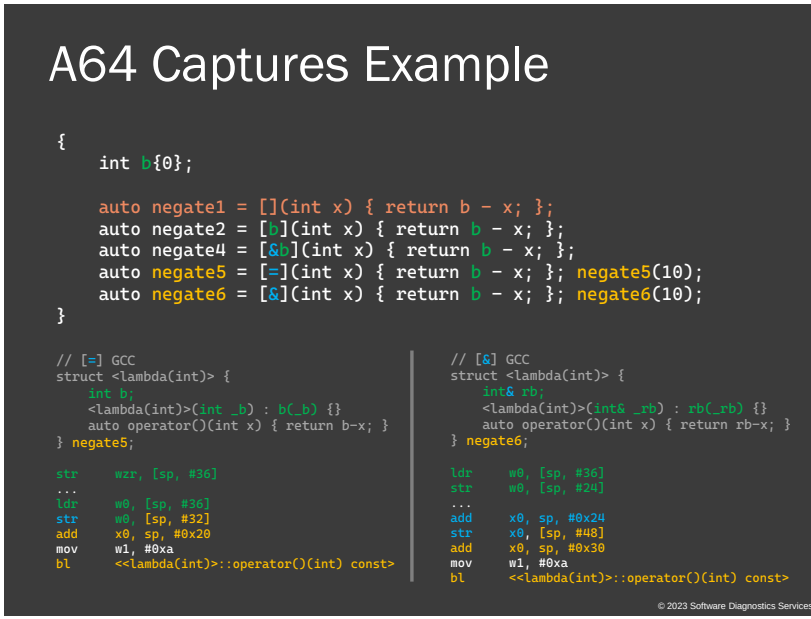
```
{
    int b{0};
    auto negate1 = [](int x) { return b - x; };
    auto negate2 = [b](int x) { return b - x; };
    auto negate4 = [&b](int x) { return b - x; };
    auto negate5 = [=](int x) { return b - x; };
    auto negate6 = [&](int x) { return b - x; };
}
```

The Clang pseudo-code examples and the corresponding x64 disassembly:

```
// [=] Clang
struct $_0 {
 int b;
 $_0(int _b) : b(_b) {}
 auto operator()(int x) { return b-x; }
} negate5;

movl   $0x0,-0x4(%rbp)
...
mov    -0x4(%rbp),%eax
mov    %eax,-0x18(%rbp)
lea    -0x18(%rbp),%rdi
mov    $0xa,%esi
callq  <$_0::operator()(int) const>
```

```
// [&] Clang
struct $_1 {
 int& rb;
 lambda_6(int& _rb) : rb(_rb) {}
 auto operator()(int x) { return rb-x; }
} negate6;

movl   $0x0,-0x4(%rbp)
...
lea    -0x4(%rbp),%rcx
mov    %rcx,-0x20(%rbp)
lea    -0x20(%rbp),%rdi
mov    $0xa,%esi
callq  <$_1::operator()(int) const>
```

A64 Captures Example

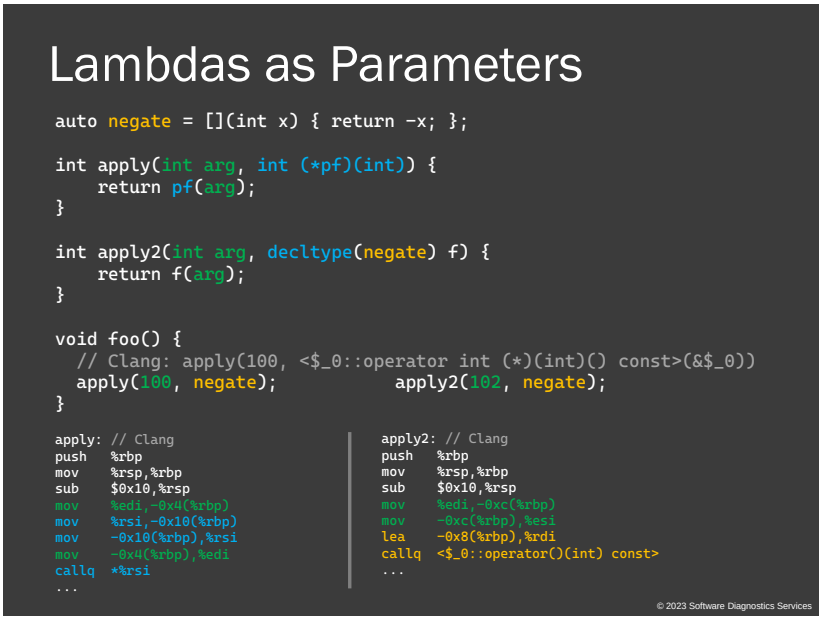


The GCC pseudo-code for the same lambdas and the corresponding ARM64 disassembly:

```
// [=] GCC
struct <lambda(int)> {
 int b;
 <lambda(int)>(int _b) : b(_b) {}
 auto operator()(int x) { return b-x; }
} negate5;

str     wzr, [sp, #36]
...
ldr     w0, [sp, #36]
str     w0, [sp, #32]
add     x0, sp, #0x20
mov     w1, #0xa
bl      <<lambda(int)>::operator()(int)
const>
```

```
// [&] GCC
struct <lambda(int)> {
 int& rb;
 <lambda(int)>(int& _rb) : rb(_rb) {}
 auto operator()(int x) { return rb-x; }
} negate6;

ldr     w0, [sp, #36]
str     w0, [sp, #24]
...
add     x0, sp, #0x24
str     x0, [sp, #48]
add     x0, sp, #0x30
mov     w1, #0xa
bl      <<lambda(int)>::operator()(int)
const>
```

Lambdas as Parameters



Lambdas can be passed as function parameters. We can use `decltype` to specify their type. If a normal function pointer is expected, then a special invoker function is internally called that takes care of the call. If the lambda type parameter is expected, the lambda function object `operator()` is called.

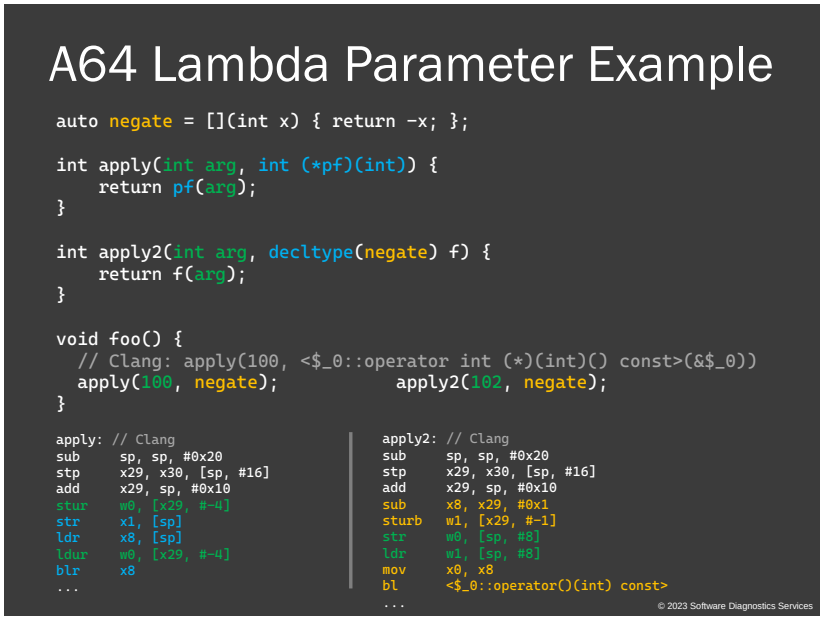
```
auto negate = [](int x) { return -x; };

int apply(int arg, int (*pf)(int)) {
    return pf(arg);
}

int apply2(int arg, decltype(negate) f) {
    return f(arg);
}

void foo() {
  // Clang: apply(100, <$_0::operator int (*)(int)() const>(&$_0))
  apply(100, negate);           apply2(102, negate);
}
```

The corresponding Clang x64 assembly language fragments:

```
apply: // Clang
push   %rbp
mov    %rsp,%rbp
sub    $0x10,%rsp
mov    %edi,-0x4(%rbp)
mov    %rsi,-0x10(%rbp)
mov    -0x10(%rbp),%rsi
mov    -0x4(%rbp),%edi
callq  *%rsi
...
```

```
apply2: // Clang
push   %rbp
mov    %rsp,%rbp
sub    $0x10,%rsp
mov    %edi,-0xc(%rbp)
mov    -0xc(%rbp),%esi
lea    -0x8(%rbp),%rdi
callq  <$_0::operator()(int) const>
...
```

A64 Lambda Parameter Example

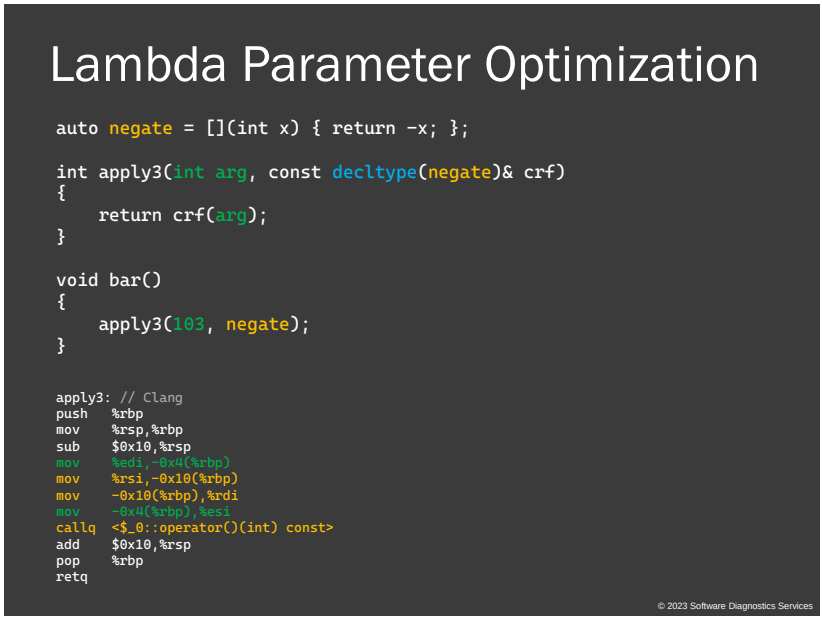


The corresponding Clang ARM64 assembly language fragments:

```
apply: // Clang
sub     sp, sp, #0x20
stp     x29, x30, [sp, #16]
add     x29, sp, #0x10
stur    w0, [x29, #-4]
str     x1, [sp]
ldr     x8, [sp]
ldur    w0, [x29, #-4]
blr     x8
...
```

```
apply2: // Clang
sub     sp, sp, #0x20
stp     x29, x30, [sp, #16]
add     x29, sp, #0x10
sub     x8, x29, #0x1
sturb   w1, [x29, #-1]
str     w0, [sp, #8]
ldr     w1, [sp, #8]
mov     x0, x8
bl      <$_0::operator()(int) const>
...
```

Lambda Parameter Optimization



If you pass lambdas by reference, there is no function object copy.

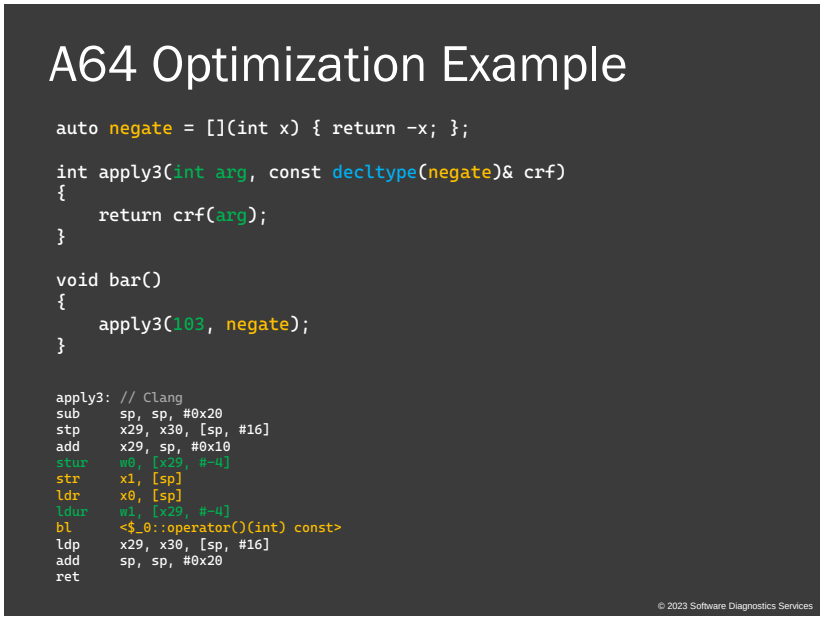
The code example and the corresponding assembly language:

```
auto negate = [](int x) { return -x; };

int apply3(int arg, const decltype(negate)& crf)
{
    return crf(arg);
}

void bar()
{
    apply3(103, negate);
}
```

In the x64 Clang disassembly, we see the function object address passed via RSI that later becomes an implicit RDI parameter for the `operator()`:

```
apply3: // Clang
push   %rbp
mov    %rsp,%rbp
sub    $0x10,%rsp
mov    %edi,-0x4(%rbp)
mov    %rsi,-0x10(%rbp)
mov    -0x10(%rbp),%rdi
mov    -0x4(%rbp),%esi
callq  <$_0::operator()(int) const>
add    $0x10,%rsp
pop    %rbp
retq
```

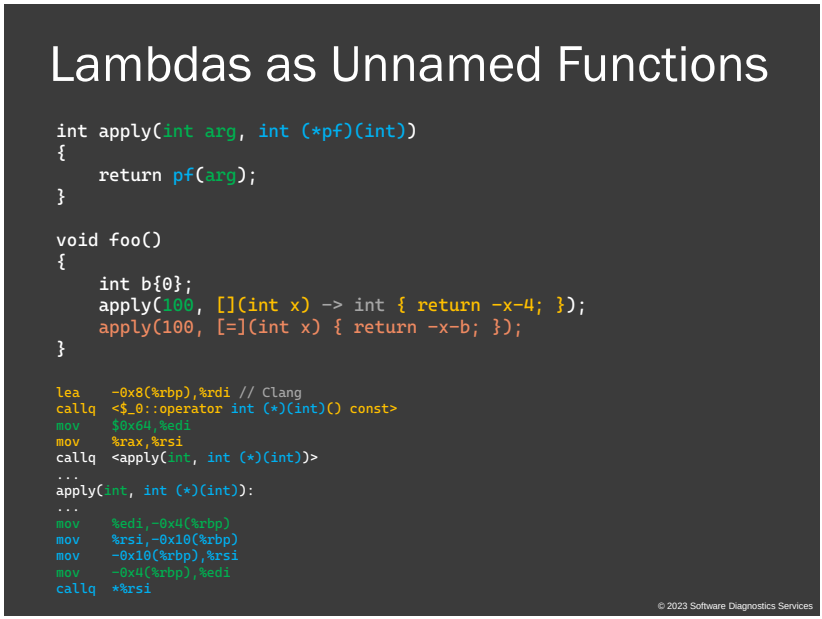
A64 Optimization Example



Clang ARM64 disassembly example:

```
apply3: // Clang
sub     sp, sp, #0x20
stp     x29, x30, [sp, #16]
add     x29, sp, #0x10
stur    w0, [x29, #-4]
str     x1, [sp]
ldr     x0, [sp]
ldur    w1, [x29, #-4]
bl      <$_0::operator()(int) const>
ldp     x29, x30, [sp, #16]
add     sp, sp, #0x20
ret
```

Lambdas as Unnamed Functions



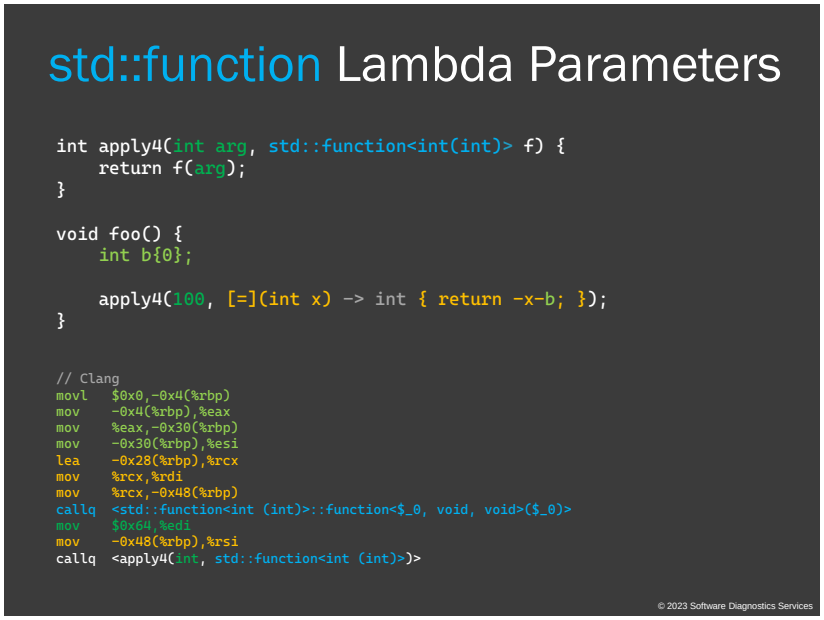
However, the most common usage of lambdas is unnamed functions in a local context. In such a case, a temporary function object is created. However, no context capture is allowed if lambdas are passed where function pointers are expected.

The code example and the corresponding Clang x64 assembly language:

```
int apply(int arg, int (*pf)(int))
{
    return pf(arg);
}

void foo()
{
    int b{0};
    apply(100, [](int x) -> int { return -x-4; });
    apply(100, [=](int x) { return -x-b; });
}
```

```
// Clang
lea     -0x8(%rbp),%rdi
callq   <$_0::operator int (*)(int)() const>
mov     $0x64,%edi
mov     %rax,%rsi     // $_0::__invoke(int)
callq   <apply(int, int (*)(int))>
...
apply(int, int (*)(int)):
...
mov     %edi,-0x4(%rbp)
mov     %rsi,-0x10(%rbp)
mov     -0x10(%rbp),%rsi
mov     -0x4(%rbp),%edi
callq   *%rsi
```

std::function Lambda Parameters



To allow passing lambdas to capture context, we can use `std::function`.

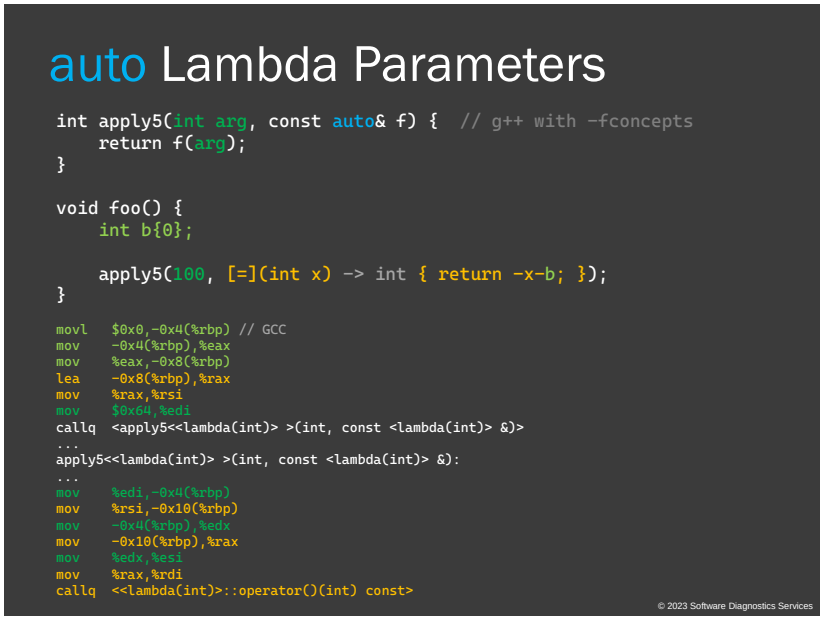
The code example and the corresponding Clang x64 assembly language:

```
int apply4(int arg, std::function<int(int)> f)
{
    return f(arg);
}

void foo()
{
    int b{0};

    apply4(100, [=](int x) -> int { return -x-b; });
}
```

In Clang x64 assembly language, we see the function object constructor that captures the context:

```
// Clang
movl   $0x0,-0x4(%rbp)
mov    -0x4(%rbp),%eax
mov    %eax,-0x30(%rbp)
mov    -0x30(%rbp),%esi
lea    -0x28(%rbp),%rcx
mov    %rcx,%rdi
mov    %rcx,-0x48(%rbp)
callq  <std::function<int (int)>::function<$_0, void, void>($_0)>
mov    $0x64,%edi
mov    -0x48(%rbp),%rsi
callq  <apply4(int, std::function<int (int)>)>
```

auto Lambda Parameters



To capture the context and allow more flexibility and efficiency, the modern way is to use `auto`.

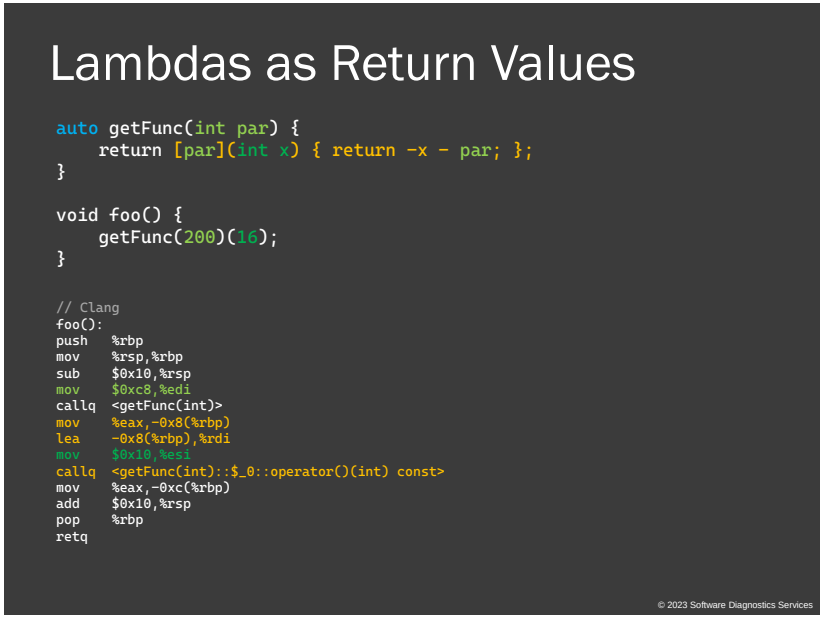
The code example:

```
int apply5(int arg, const auto& f) // g++ with -fconcepts
{
    return f(arg);
}

void foo()
{
    int b{0};

    apply5(100, [=](int x) -> int { return -x-b; });
}
```

In the corresponding GCC x64 disassembly, we see all necessary functions are generated automatically:

```
movl   $0x0,-0x4(%rbp) // GCC
mov    -0x4(%rbp),%eax
mov    %eax,-0x8(%rbp)
lea    -0x8(%rbp),%rax
mov    %rax,%rsi
mov    $0x64,%edi
callq  <apply5<<lambda(int)> >(int, const <lambda(int)> &)>
...
apply5<<lambda(int)> >(int, const <lambda(int)> &):
...
mov    %edi,-0x4(%rbp)
mov    %rsi,-0x10(%rbp)
mov    -0x4(%rbp),%edx
mov    -0x10(%rbp),%rax
mov    %edx,%esi
mov    %rax,%rdi
callq  <<lambda(int)>::operator()(int) const>
```

Lambdas as Return Values



It is possible to return lambdas and thus mimic the so-called currying feature of functional programming.

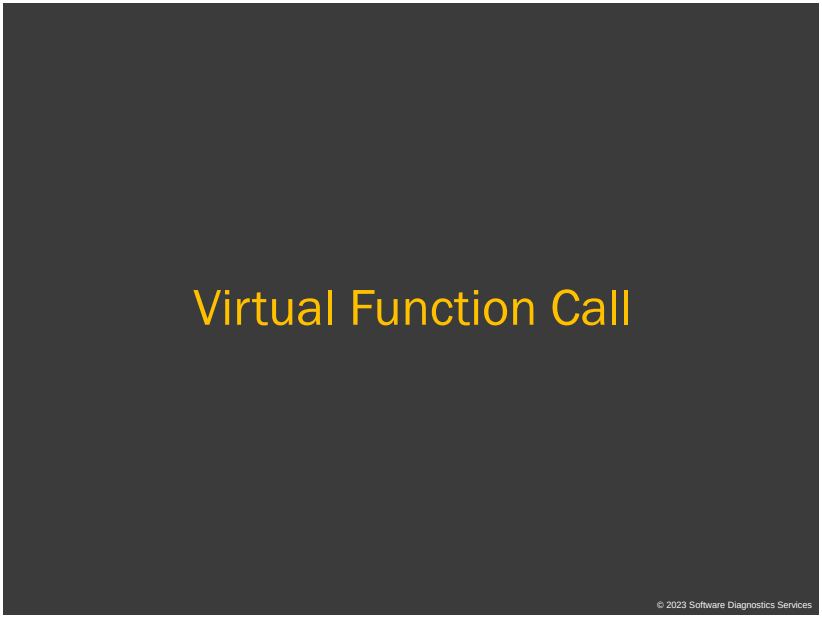
The code example and the corresponding Clang x64 assembly language:

```
auto getFunc(int par)
{
    return [par](int x) { return -x - par; };
}

void foo()
{
    getFunc(200)(16);
}
```

The disassembly shows that we return the underlying function object:

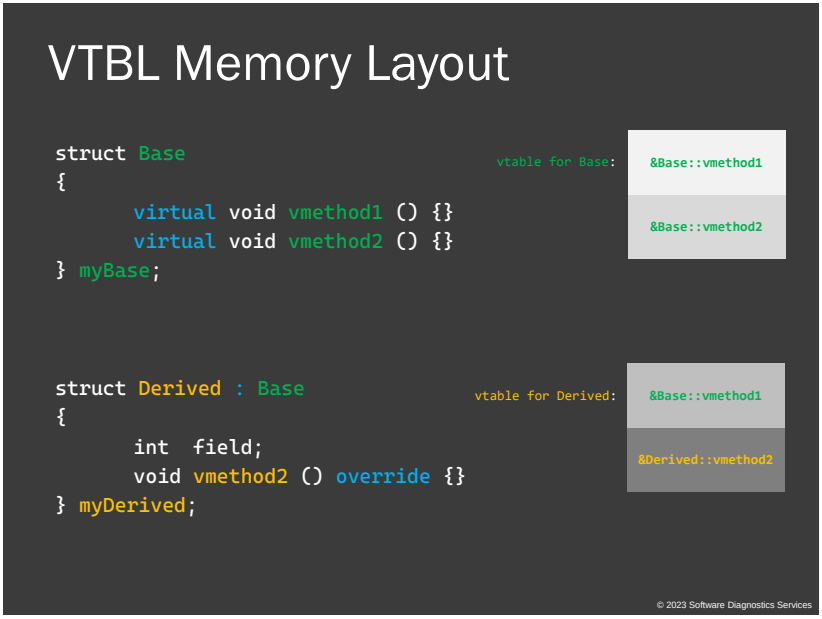
```
// Clang
foo():
push   %rbp
mov    %rsp,%rbp
sub    $0x10,%rsp
mov    $0xc8,%edi
callq  <getFunc(int)>
mov    %eax,-0x8(%rbp)
lea    -0x8(%rbp),%rdi
mov    $0x10,%esi
callq  <getFunc(int)::$_0::operator()(int) const>
mov    %eax,-0xc(%rbp)
add    $0x10,%rsp
pop    %rbp
retq
```

Virtual Function Call



This section provides an overview of virtual function calls in C++.

VTBL Memory Layout

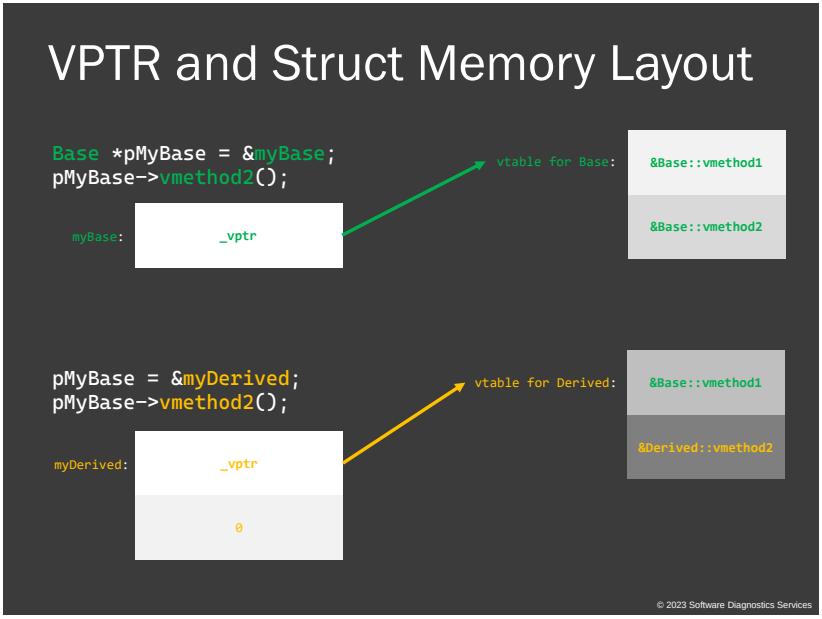


These virtual function calls are implemented uniformly by having a specific virtual function table (vtbl or vtable) for each structure where the addresses of the base structure methods are replaced with those of the derived structure methods, if any.

The code examples corresponding to the memory diagram:

```
struct Base
{
    virtual void vmethod1 () {}
    virtual void vmethod2 () {}
} myBase;

struct Derived : Base
{
    int  field;
    void vmethod2 () override {}
} myDerived;
```

VPTR and Struct Memory Layout

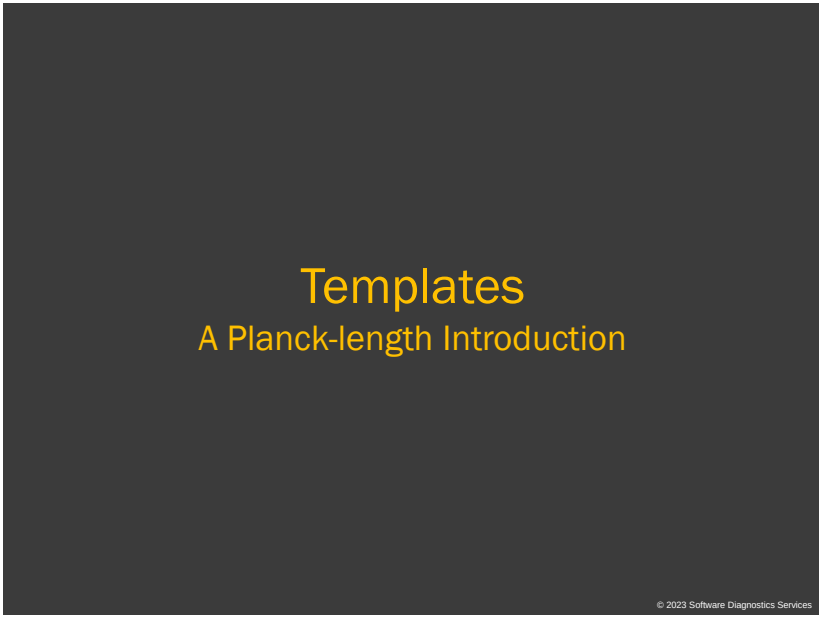


Every object whose structure has virtual methods has an implicit virtual function table pointer (`_vptr`) as its first member containing an address of the corresponding structure virtual functions table. Therefore, each virtual function call from a base structure pointer is a type-independent call where the target function address is easily calculated based on the address of the virtual function table and virtual function offset.

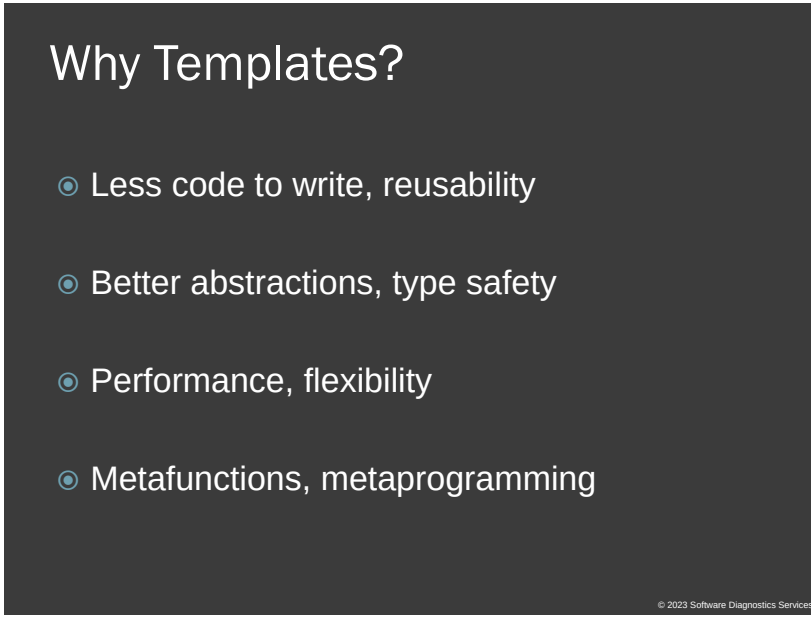
The code examples corresponding to the memory diagram:

```
Base *pMyBase = &myBase;
pMyBase->vmethod2();

pMyBase = &myDerived;
pMyBase->vmethod2();
```

Templates: A Planck-length Introduction

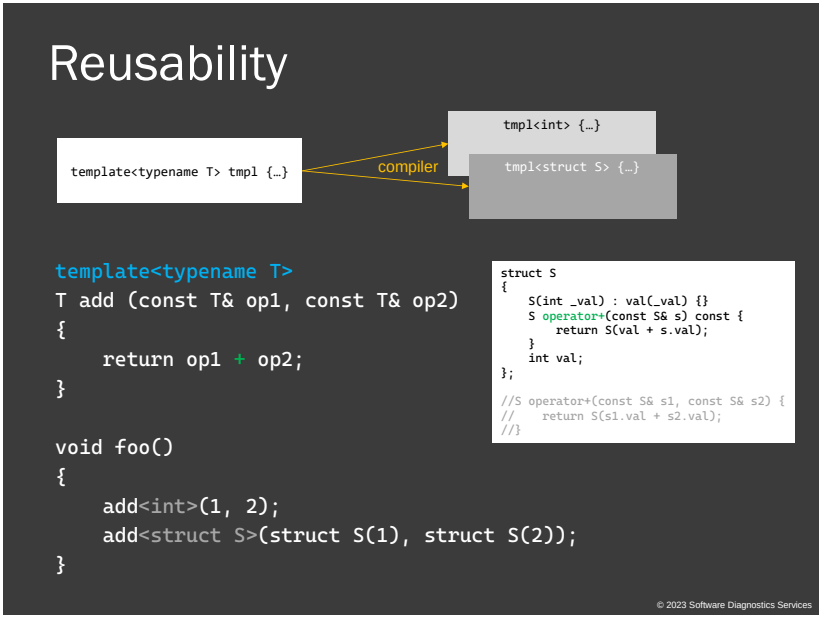


C++ templates and template metaprogramming are a vast universe. We only cover a Planck-length distance in this introduction.



In the following slides, we briefly cover various “why” aspects.

Reusability



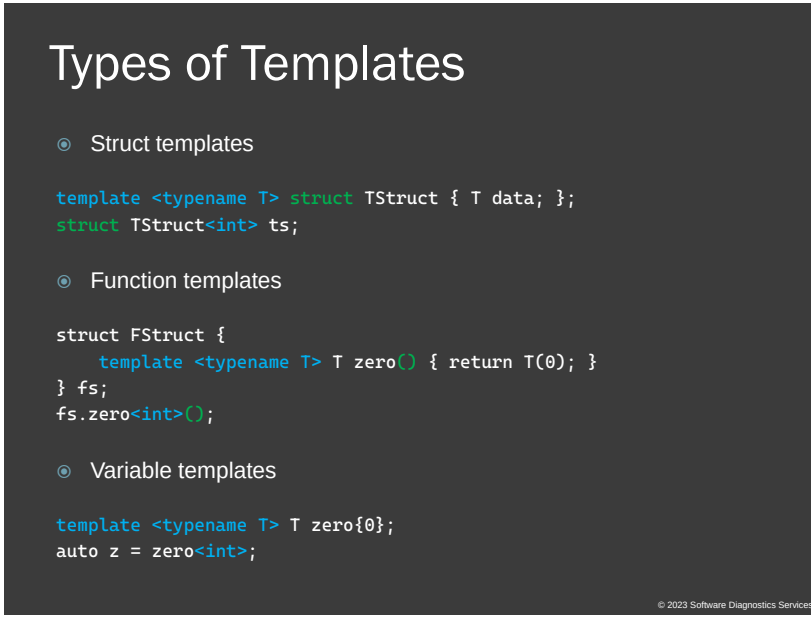
Templates allow us to write less code with higher abstractions, delegating implementation details to a compiler. The compiler checks that template arguments are compatible, for example, that they implement the required operations and methods.

The code example:

```
template<typename T>
T add (const T& op1, const T& op2)
{
    return op1 + op2;
}

void foo()
{
    add<int>(1, 2);
    add<struct S>(struct S(1), struct S(2));
}
```

```
struct S
{
    S(int _val) : val(_val) {}
    S operator+(const S& s) const
    {
        return S(val + s.val);
    }
    int val;
};

// S operator+(const S& s1, const S& s2)
// {
//    return S(s1.val + s2.val);
// }
```

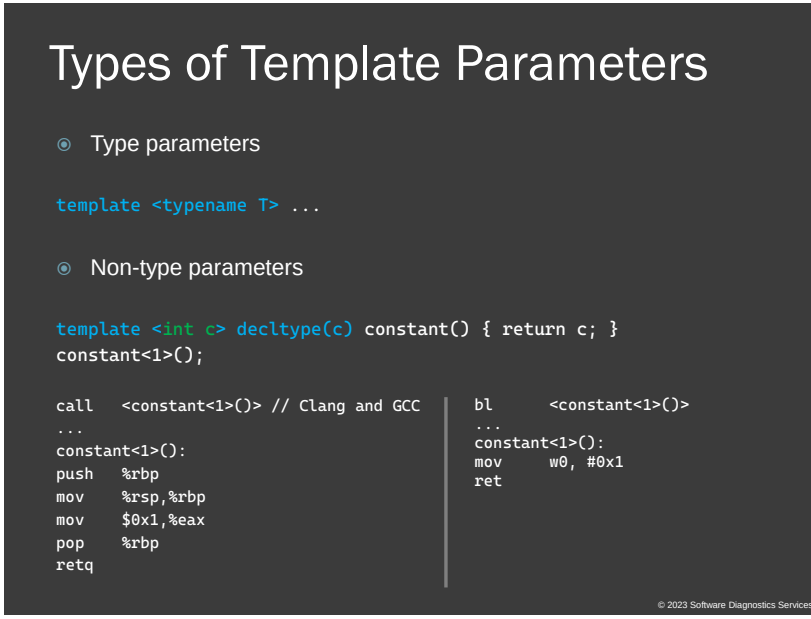
Types of Templates



The following code provides examples for struct, function, and (recent) variable template categories:

```
template <typename T> struct TStruct { T data; };
struct TStruct<int> ts;

struct FStruct
{
    template <typename T> T zero() { return T(0); }
} fs;
fs.zero<int>();

template <typename T> T zero{0};
auto z = zero<int>;
```

Types of Template Parameters



There are two categories of template parameters: type and non-type.

The code example and the corresponding assembly language:

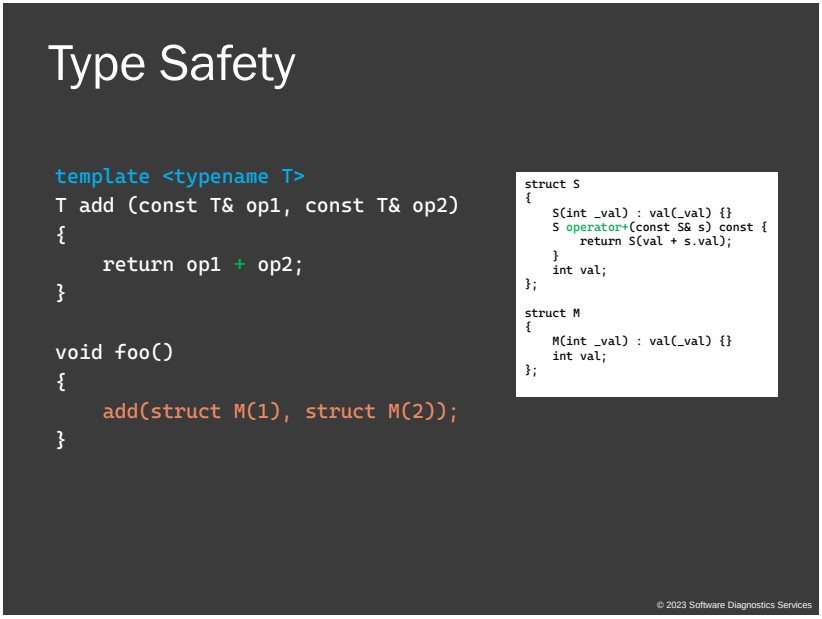
```
template <int c> decltype(c) constant() { return c; }
constant<1>();
```

Non-type template parameters allow the generation of very compact code with a distinct purpose, as shown in x64 and ARM64 disassembly:

```
call   <constant<1>()> // Clang and GCC
...
constant<1>():
push   %rbp
mov    %rsp,%rbp
mov    $0x1,%eax
pop    %rbp
retq
```

```
bl      <constant<1>()>
...
constant<1>():
mov     w0, #0x1
ret
```

Type Safety



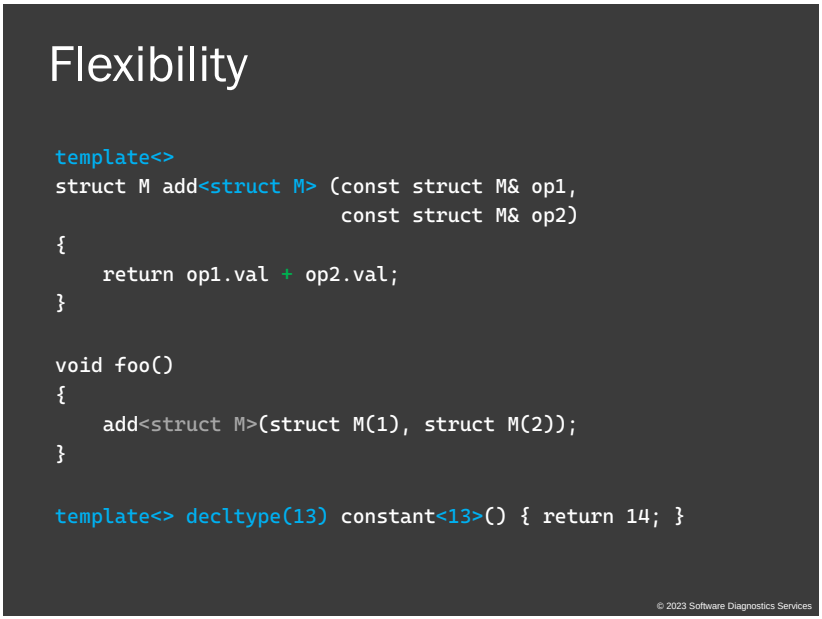
Compared to pointer casting, the general template code enforces type safety by checking the required operations and compatible types.

The code example:

```
struct S
{
    S(int _val) : val(_val) {}
    S operator+(const S& s) const {
        return S(val + s.val);
    }
    int val;
};

struct M
{
    M(int _val) : val(_val) {}
    int val;
};
```

```
template <typename T>
T add (const T& op1, const T& op2)
{
    return op1 + op2;
}

void foo()
{
    add(struct M(1), struct M(2));
}
```

Flexibility

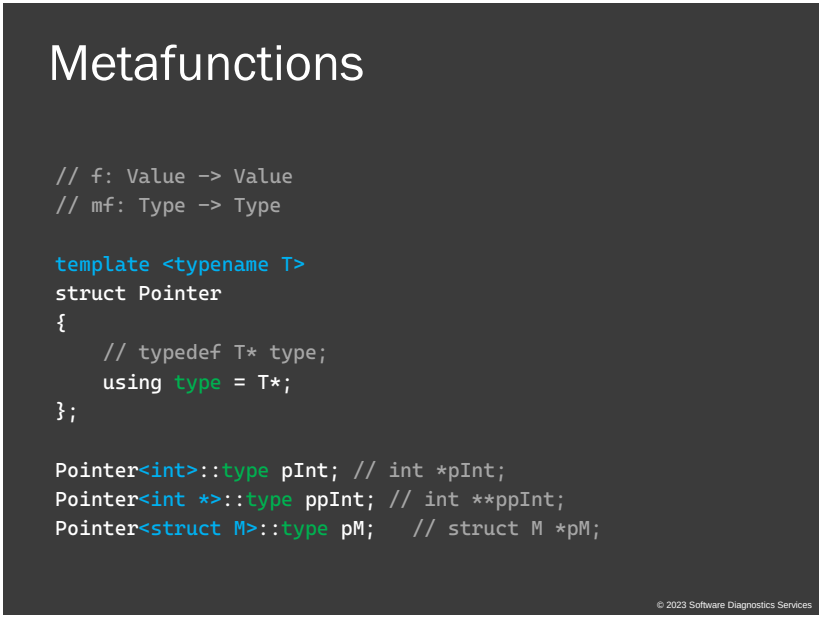


It is also possible to specialize template definitions for specific types for performance and other reasons.

The code example:

```
template<>
struct M add<struct M> (const struct M& op1,
                        const struct M& op2)
{
    return op1.val + op2.val;
}

void foo()
{
    add<struct M>(struct M(1), struct M(2));
}

template<> decltype(13) constant<13>() { return 14; }
```

Metafunctions

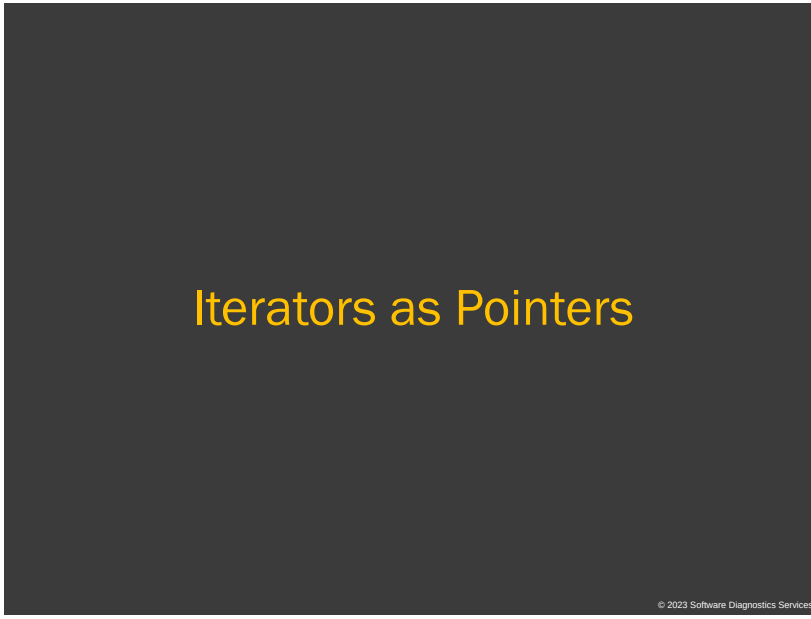


Metafunctions transform type: they take types as parameters and return types as output.

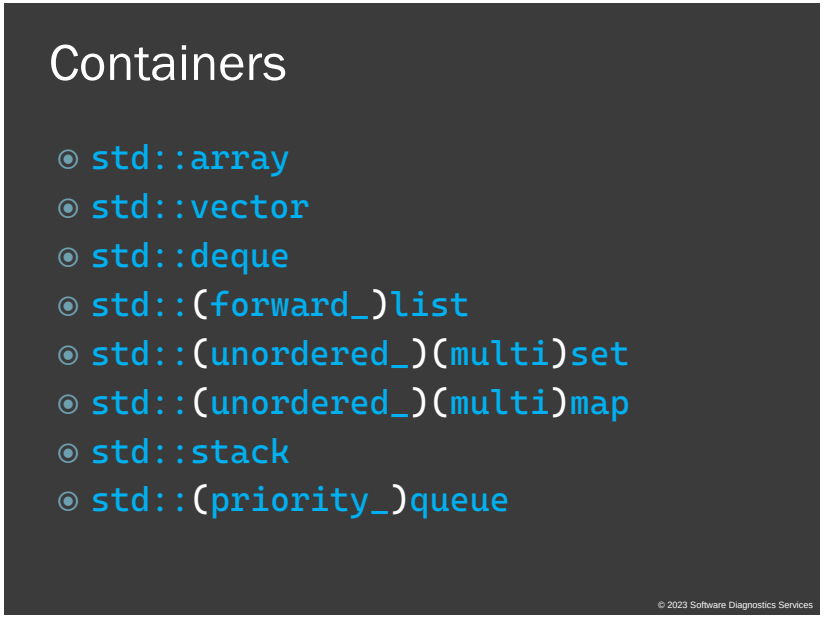
The code example:

```
// f: Value -> Value
// mf: Type -> Type

template <typename T>
struct Pointer
{
    // typedef T* type;
    using type = T*;
};

Pointer<int>::type pInt; // int *pInt;
Pointer<int *>::type ppInt; // int **ppInt;
Pointer<struct M>::type pM;   // struct M *pM;
```

Iterators as Pointers

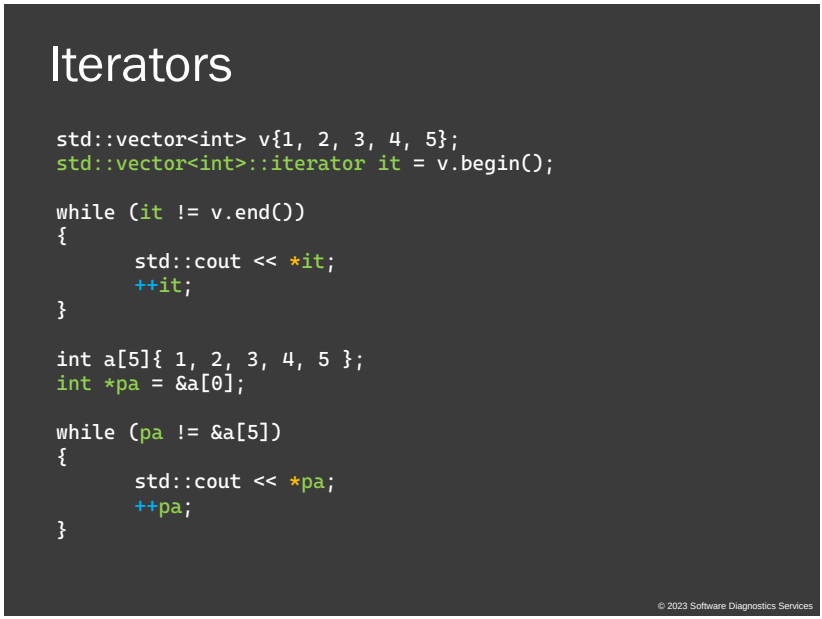


Now, we take a bird's eye view of standard library containers, iterators, and algorithms from a pointer perspective.



There are many container types in the standard C++ library (formerly called STL, Standard Template Library). In this course edition, we don't delve into their specifics. We hope the names intuitively suggest their semantics.

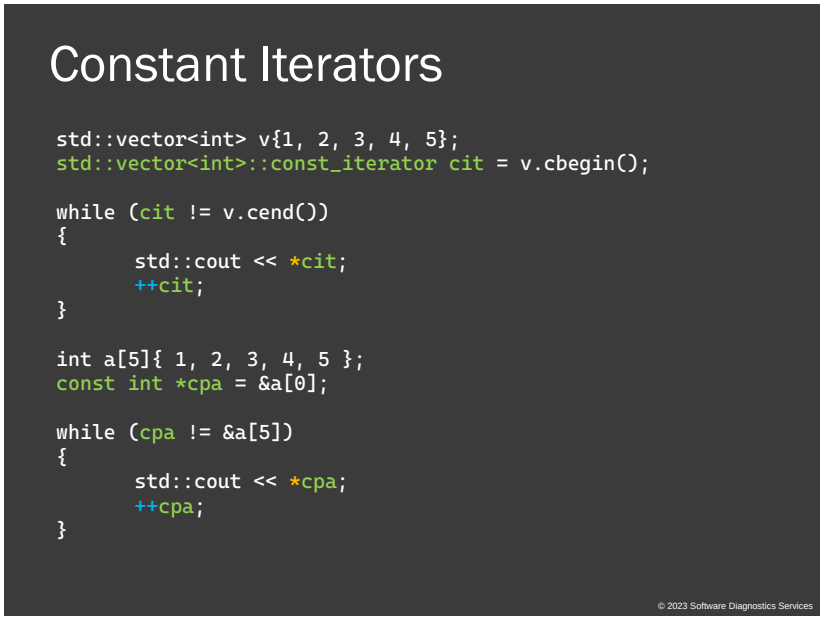
Iterators



An iterator is a pointer abstraction. You can move it (depending on container semantics) like a pointer increment/decrement, and you can dereference it like a pointer to get a value:

```
std::vector<int> v{1, 2, 3, 4, 5};
std::vector<int>::iterator it = v.begin();

while (it != v.end())
{
    std::cout << *it;
    ++it;
}

int a[5]{ 1, 2, 3, 4, 5 };
int *pa = &a[0];

while (pa != &a[5])
{
    std::cout << *pa;
    ++pa;
}
```

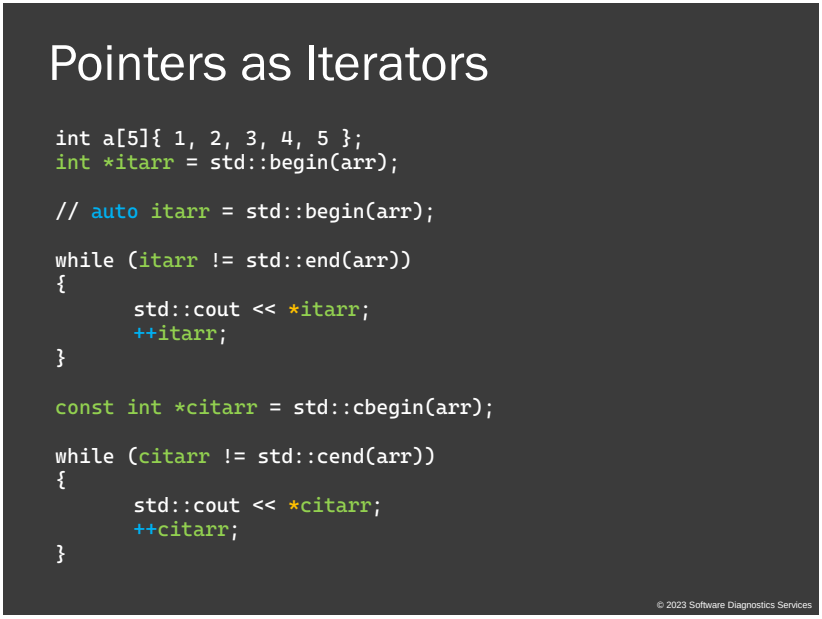
Constant Iterators



Like a pointer to constant values, there are constant iterators:

```
std::vector<int> v{1, 2, 3, 4, 5};
std::vector<int>::const_iterator cit = v.cbegin();

while (cit != v.cend())
{
      std::cout << *cit;
      ++cit;
}

int a[5]{ 1, 2, 3, 4, 5 };
const int *cpa = &a[0];

while (cpa != &a[5])
{
      std::cout << *cpa;
      ++cpa;
}
```

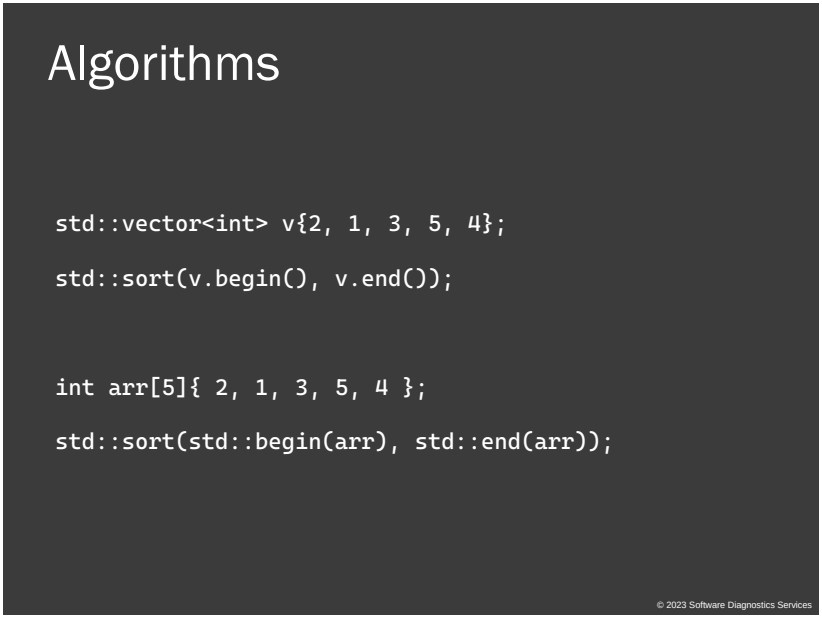
Pointers as Iterators



Pointers can also be considered as iterators:

```
int a[5]{ 1, 2, 3, 4, 5 };
int *itarr = std::begin(arr);

// auto itarr = std::begin(arr);

while (itarr != std::end(arr))
{
	std::cout << *itarr;
	++itarr;
}

const int *citarr = std::cbegin(arr);

while (citarr != std::cend(arr))
{
	std::cout << *citarr;
	++citarr;
}
```

Algorithms



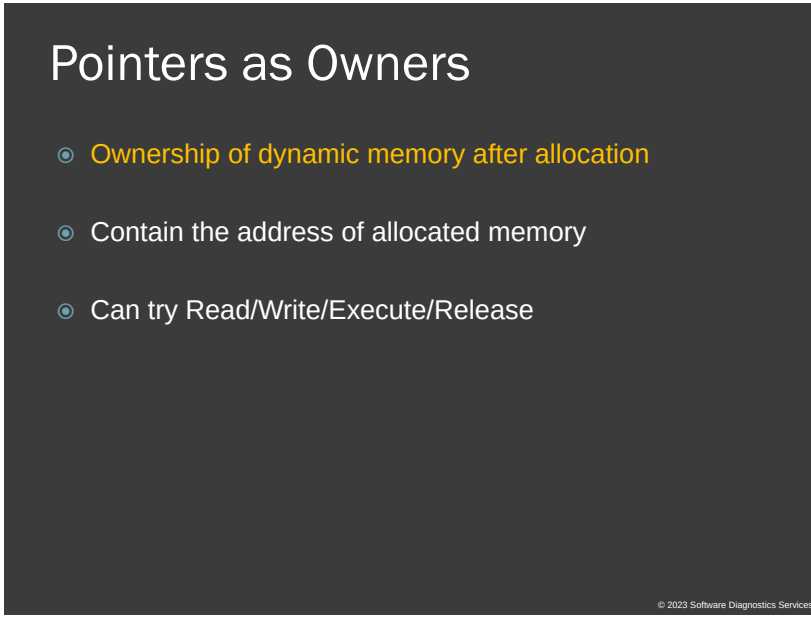
Both iterators and containers and pointers as iterators and arrays can be used with the C++ standard library algorithms.

The code examples:

```
std::vector<int> v{2, 1, 3, 5, 4};
std::sort(v.begin(), v.end());

int arr[5]{ 2, 1, 3, 5, 4 };
std::sort(std::begin(arr), std::end(arr));
```

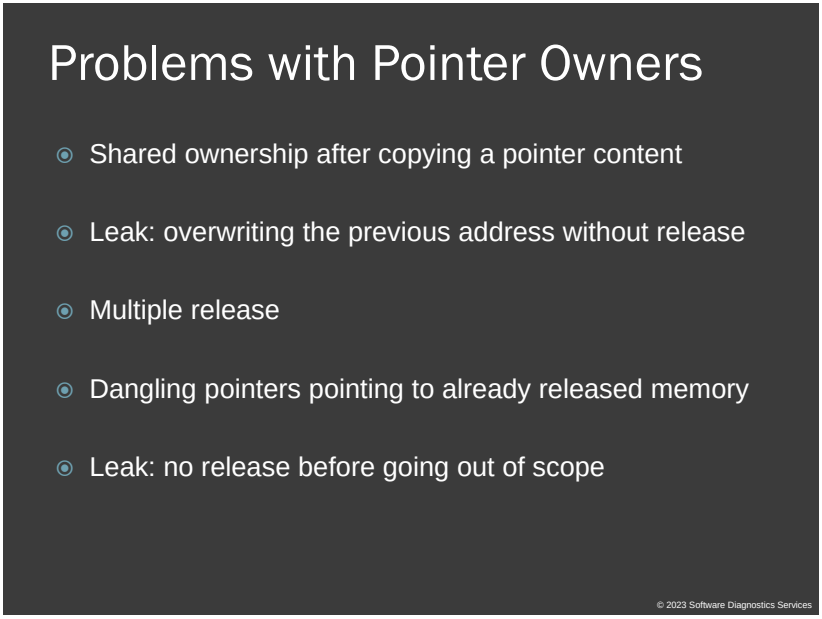
Memory Ownership

We now look at common memory ownership problems and see how they are resolved in modern C++.



Pointers can be considered owners of dynamically allocated memory since they contain the address, and that memory can be accessed through them.

Problems with Pointer Owners

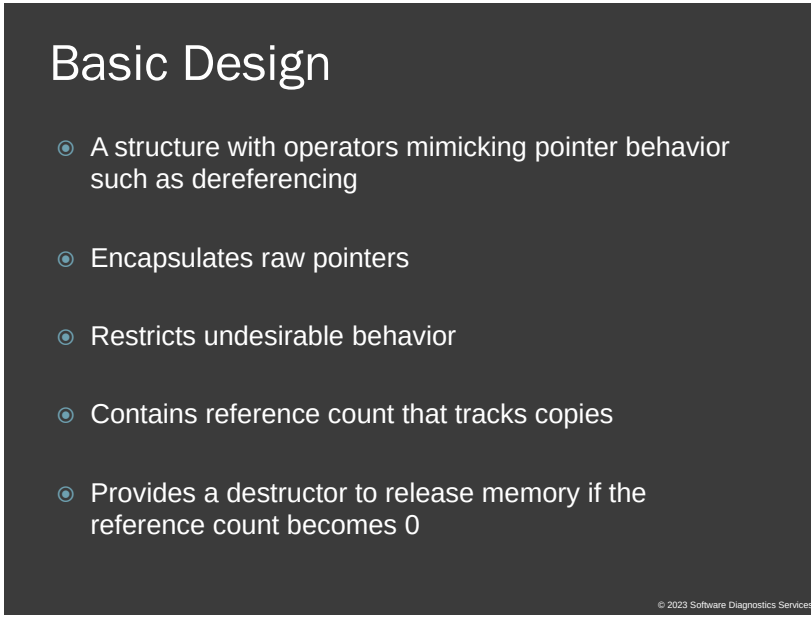


However, manual pointer usage is prone to multiple errors due to possible shared ownership, crashes due to possible multiple releases, and memory leaks due to dangling pointers and going out of scope.

Smart Pointers

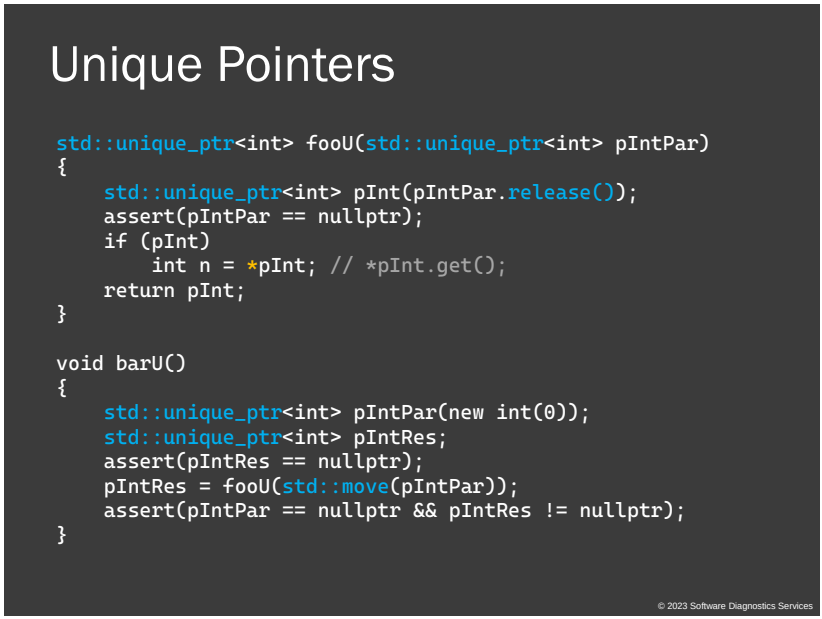
To solve memory ownership problems, modern C++ included several kinds of smart pointers in its standard library.

Basic Design



A smart pointer should include functionality similar to raw pointers for seamless use during the refactoring of legacy code and simultaneously eliminate most, if not all, problems with raw pointer usage.

Unique Pointers

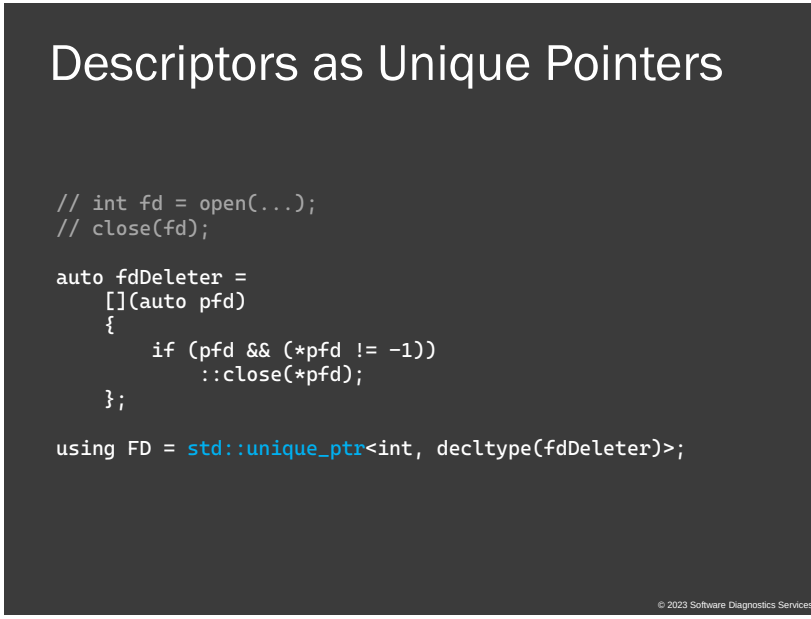


For smart pointers without sharing functionality, the `unique_ptr` should be used. Copying must transfer ownership, making the source `nullptr`.

The code example:

```
std:: <int> fooU(std::unique_ptr<int> pIntPar)
{
    std::unique_ptr<int> pInt(pIntPar.release());
    assert(pIntPar == nullptr);
    if (pInt)
        int n = *pInt; // *pInt.get();
    return pInt;
}

void barU() {
    std::unique_ptr<int> pIntPar(new int(0));
    std::unique_ptr<int> pIntRes;
    assert(pIntRes == nullptr);
    pIntRes = fooU(std::move(pIntPar));
    assert(pIntPar == nullptr && pIntRes != nullptr);
}
```

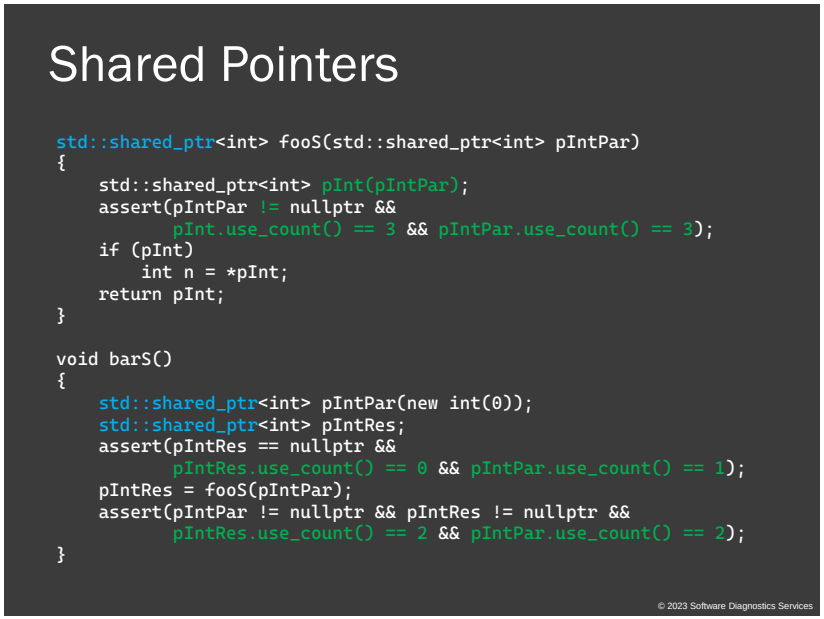
Descriptors as Unique Pointers



File descriptors are good candidates for unique pointers, but they should be supplied with a custom deletion mechanism.

The code example:

```
// int fd = open(...);
// close(fd);

auto fdDeleter =
    [](auto pfd)
    {
        if (pfd && (*pfd != -1))
            ::close(*pfd);
    };

using FD = std::unique_ptr<int, decltype(fdDeleter)>;
```



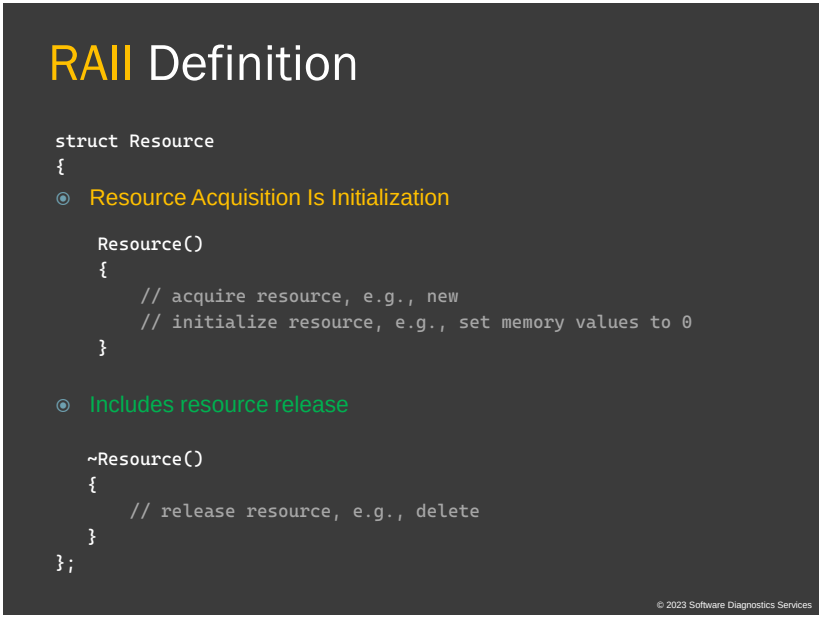
If we want to freely copy pointers around with all new copies pointing to the same memory, then the `shared_ptr` is our choice:

```
std::shared_ptr<int> fooS(std::shared_ptr<int> pIntPar)
{
    std::shared_ptr<int> pInt(pIntPar);
    assert(pIntPar != nullptr &&
           pInt.use_count() == 3 && pIntPar.use_count() == 3);
    if (pInt)
        int n = *pInt;
    return pInt;
}

void barS()
{
    std::shared_ptr<int> pIntPar(new int(0));
    std::shared_ptr<int> pIntRes;
    assert(pIntRes == nullptr &&
           pIntRes.use_count() == 0 && pIntPar.use_count() == 1);
    pIntRes = fooS(pIntPar);
    assert(pIntPar != nullptr && pIntRes != nullptr &&
           pIntRes.use_count() == 2 && pIntPar.use_count() == 2);
}
```

RAII

Finally, the RAII idiom is specifically about managing resources, including memory,

RAII Definition

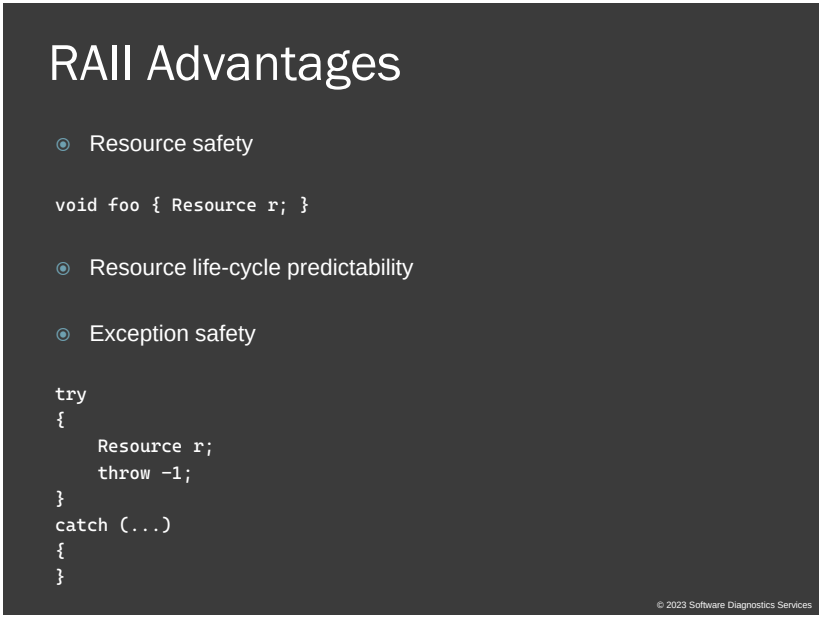


An RAII structure encapsulates simultaneous “atomic” resource acquisition and initialization in its constructors and includes resource release logic in its destructor.

The code example:

```
struct Resource
{
    Resource()
    {
        // acquire resource, e.g., new
        // initialize resource, e.g., set memory values to 0
    }

   ~Resource()
   {
       // release resource, e.g., delete
   }
};
```

RAII Advantages



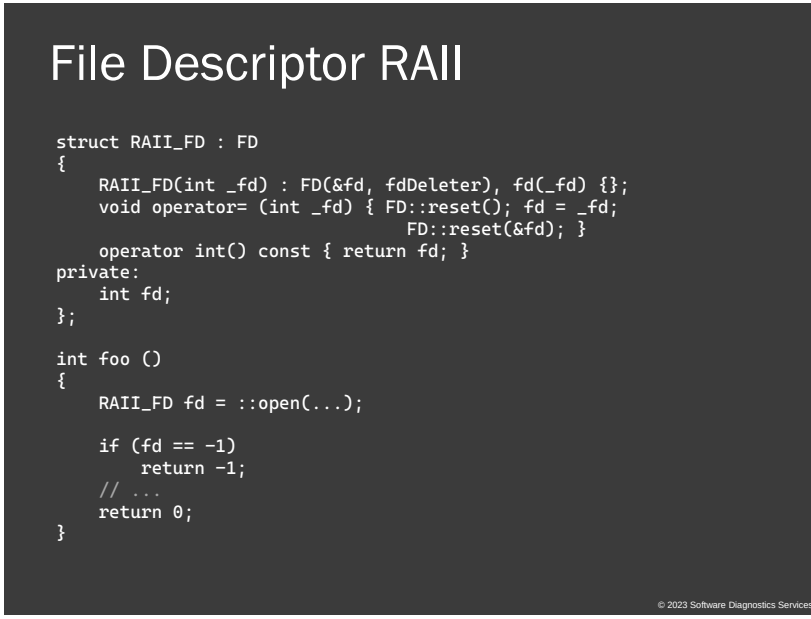
There are several advantages to using the RAII idiom. When going out of scope, it automatically releases a resource due to a called destructor:

```
void foo { Resource r; }
```

In the presence of exceptions, the destructor releases the acquired resource automatically. All these contribute to the predictable resource life cycle.

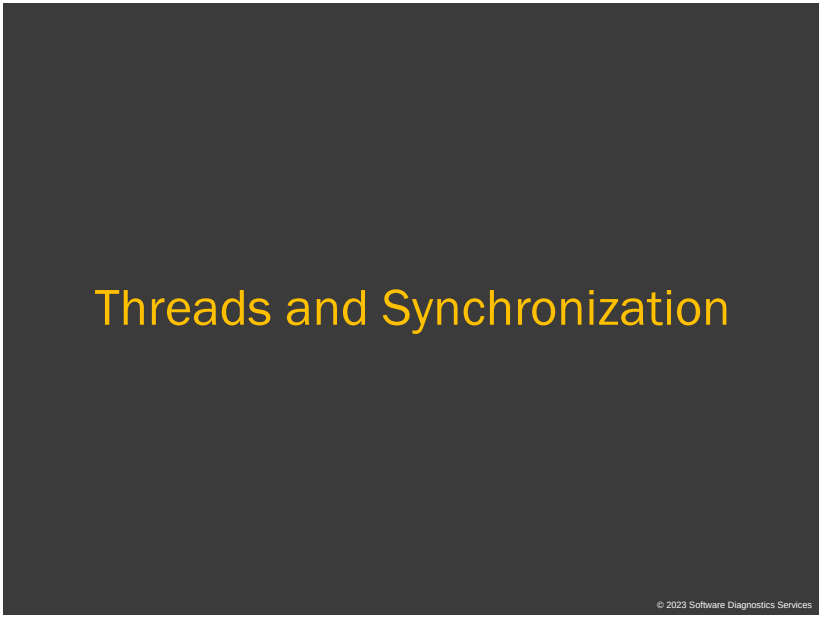
```
try
{
    Resource r;
    throw -1;
}
catch (...)
{
}
```

File Descriptor RAII

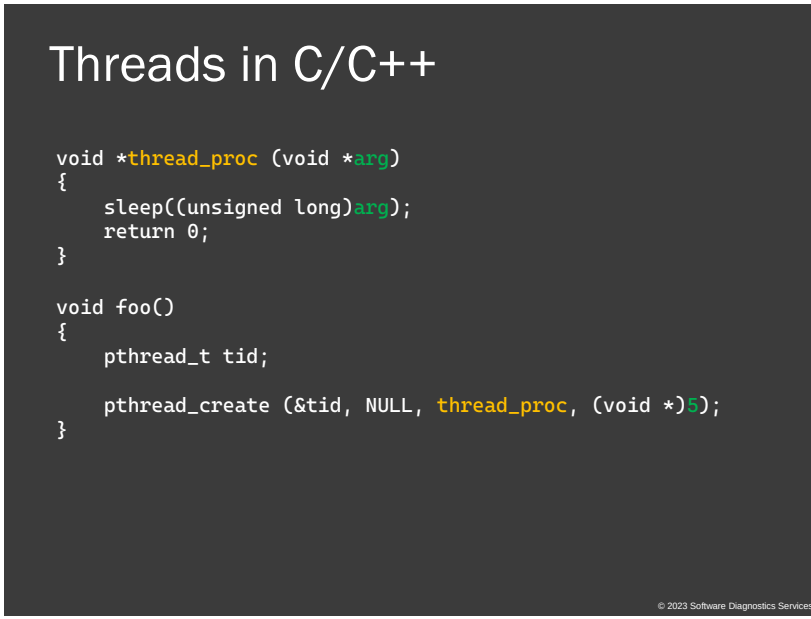


The code example of encapsulating file descriptors using the RAII idiom. We reuse the FD type from the previous *Descriptors as Unique Pointers* slide:

```
struct RAII_FD : FD
{
    RAII_FD(int _fd) : FD(&fd, fdDeleter), fd(_fd) {};
    void operator= (int _fd) { FD::reset(); fd = _fd;
                               FD::reset(&fd); }
    operator int() const { return fd; }
private:
    int fd;
};

int foo ()
{
    RAII_FD fd = ::open(...);
    if (fd == -1)
        return -1;
    // ...
    return 0;
}
```



Our final section in this edition is about threads and synchronization in C++.

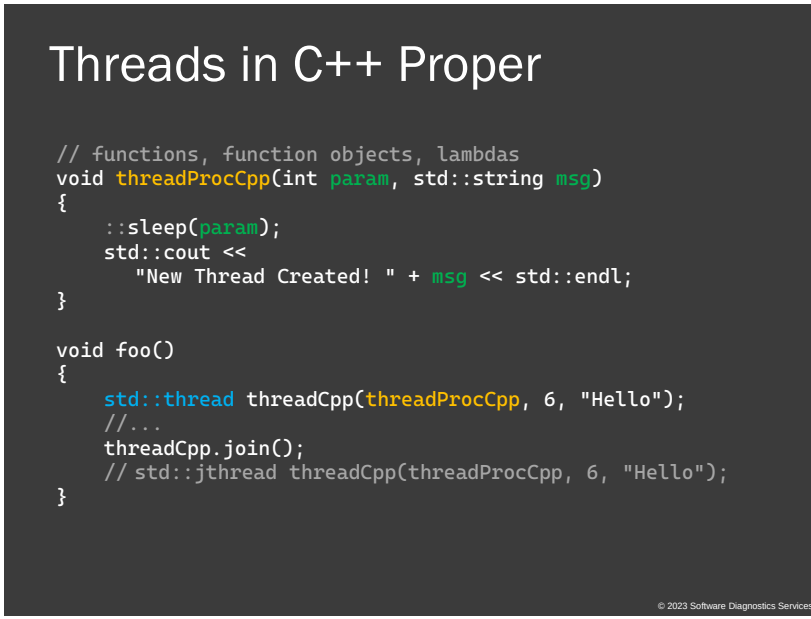


Traditional C/C++ threading using raw pthread API is limited. Passing any parameter type and casting it is error-prone.

The code example:

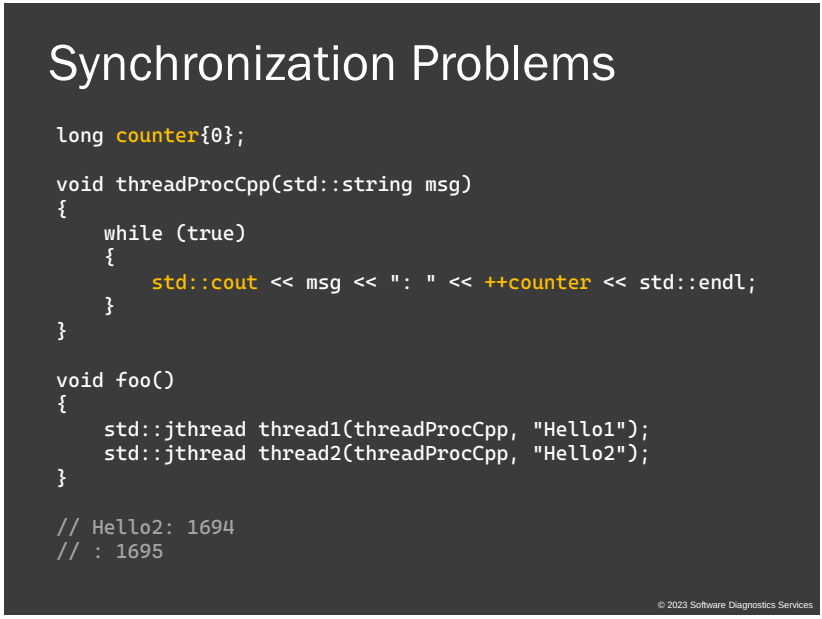
```
void *thread_proc (void *arg)
{
    sleep((unsigned long)arg);
    return 0;
}

void foo()
{
    pthread_t tid;
    pthread_create (&tid, NULL, thread_proc, (void *)5);
}
```

Threads in C++ Proper



The proper C++ standard library `thread` allows the utilization of the power of modern C++ abstractions. Please also note the existence of j`thread` that combines both thread creation and `join`:

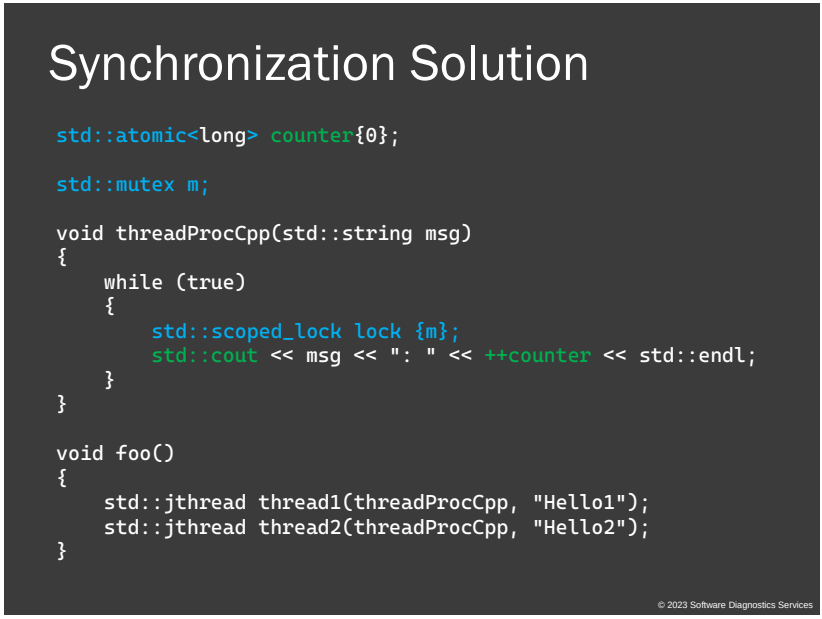
```
// functions, function objects, lambdas
void threadProcCpp(int param, std::string msg)
{
    ::sleep(param);
    std::cout <<
       "New Thread Created! " + msg << std::endl;
}

void foo()
{
    std::thread threadCpp(threadProcCpp, 6, "Hello");
    //...
    threadCpp.join();
    // std::jthread threadCpp(threadProcCpp, 6, "Hello");
}
```



In the code example, the output from `<<` operators from different threads is mixed and looks like garbage:

```
long counter{0};

void threadProcCpp(std::string msg)
{
    while (true)
    {
        std::cout << msg << ": " << ++counter << std::endl;
    }
}

void foo()
{
    std::jthread thread1(threadProcCpp, "Hello1");
    std::jthread thread2(threadProcCpp, "Hello2");
}

// Hello2: 1694
// : 1695
```

Synchronization Solution

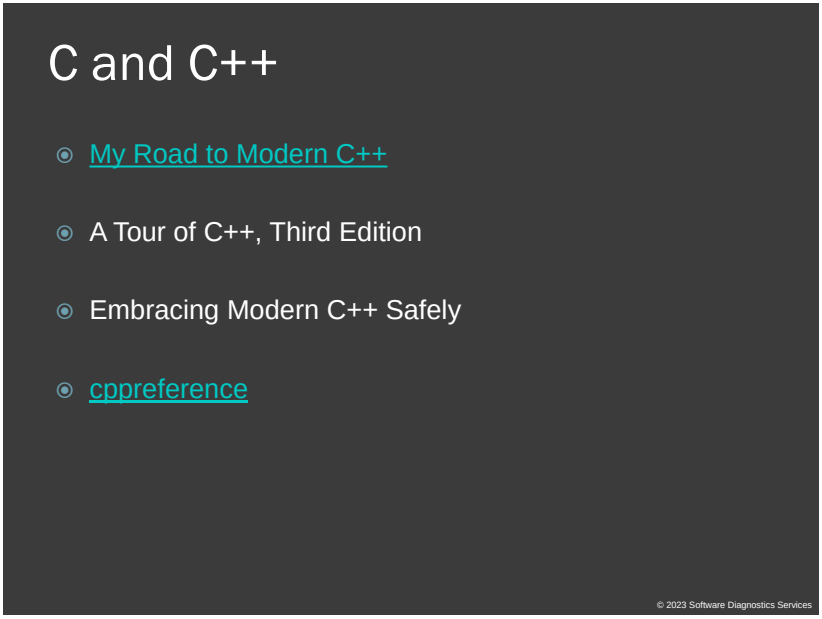


The following code example solves the data race problem by guarding access via `scoped_lock` that is implemented using `mutex`:

```
std::atomic<long> counter{0};

std::mutex m;

void threadProcCpp(std::string msg)
{
    while (true)
    {
        std::scoped_lock lock {m};
        std::cout << msg << ": " << ++counter << std::endl;
    }
}

void foo()
{
    std::jthread thread1(threadProcCpp, "Hello1");
    std::jthread thread2(threadProcCpp, "Hello2");
}
```

Resources

Now, I have a few slides about references and resources for further reading.



My reading list up to C++17:

My Road to Modern C++
https://www.linkedin.com/pulse/my-road-modern-c-dmitry-vostokov/

cppreference
https://en.cppreference.com/w/

Two recent books are also recommended:

A Tour of C++, Third Edition
Embracing Modern C++ Safely

Training (Linux C and C++)

Additional training courses that use Linux C and C++:

Accelerated Linux Core Dump Analysis, Third Edition
https://www.patterndiagnostics.com/accelerated-linux-core-dump-analysis-book

Accelerated Linux Debugging[4]
https://www.patterndiagnostics.com/accelerated-linux-debugging-4d

Accelerated Linux Disassembly, Reconstruction, and Reversing, Second Edition
https://www.patterndiagnostics.com/accelerated-linux-disassembly-reconstruction-reversing-book

Accelerated Linux API for Software Diagnostics
https://www.patterndiagnostics.com/accelerated-linux-api-book