GARMIN

{ Back to Basics

Casting

Or: how to subvert the type system

Brian Ruth (he/him) brian.ruth@garmin.com

An Introduction

```
struct region { int size; };

void init_region(char* backing_buffer, size_t buffer_size ) {
  if(buffer_size < current_region.size) {
    LOG(“Buffer size too small”);
    return;
  }
//other init code
}
```

An Introduction

```
struct region { int size; };

void init_region(char* backing_buffer, size_t buffer_size ) {
  if(buffer_size < current_region.size) {
    LOG(“Buffer size too small”);
    return;
  }
//other init code
}
```

```
<source>:13:18: warning: comparison of integer expressions of different
signedness: 'size_t' {aka 'long unsigned int'} and 'int' [-Wsign-compare]
   13 |   if(buffer_size < current_region.size) {
      |      ~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~
```

An Introduction

```
struct region { int size; };

void init_region(char* backing_buffer, size_t buffer_size ) {
  if(buffer_size < (size_t)current_region.size) {
    LOG(“Buffer size too small”);
    return;
  }
//other init code
}
```

An Introduction

```
struct region {
  constexpr int INVALID_SIZE = -1;
  int size = INVALID_SIZE;
};

void init_region(char* backing_buffer, size_t buffer_size ) {
  if(buffer_size < (size_t)current_region.size) {
    LOG("Buffer size too small");
    return;
  }
//other init code
}
```

An Introduction

```
struct region {
  constexpr int INVALID_SIZE = -1;
  int size = INVALID_SIZE;
};

void init_region(char* backing_buffer, size_t buffer_size ) {
  if(buffer_size < (size_t)current_region.size) {
    LOG(“Buffer size too small”);
    return;
  }
//other init code
}
```

Turns INVALID_SIZE into
a very large number

“In all cases, it would be better if the cast - new or old - could be eliminated”

–Bjarne Stroustrup

The Design and Evolution of C++

Why do we need casts?

C++ is a statically typed language

Casts allow us to...

1. Work with raw memory
2. Navigate inheritance hierarchy

Why do we need casts?

What is a Type?

0x43	0x50	0x50	0x43	0x4F	0x4E	0x21	0x00

What is a Type?

0x43	0x50	0x50	0x43	0x4F	0x4E	0x21	0x00

Type	Value*
`int[2]`	`{1129336899, 2182735}`

*64-bit MSVC v142

What is a Type?

0x43	0x50	0x50	0x43	0x4F	0x4E	0x21	0x00

Type	Value*
int[2]	{1129336899, 2182735}
double	4.81336e-308

*64-bit MSVC v142

What is a Type?

0x43	0x50	0x50	0x43	0x4F	0x4E	0x21	0x00

Type	Value*
int[2]	{1129336899, 2182735}
double	4.81336e-308
float[2]	{208.314, 3.05866e-39}

*64-bit MSVC v142

What is a Type?

0x43	0x50	0x50	0x43	0x4F	0x4E	0x21	0x00

Type	Value*
`int[2]`	`{1129336899, 2182735}`
`double`	`4.81336e-308`
`float[2]`	`{208.314, 3.05866e-39}`
`size_t`	`9374776570171459`

*64-bit MSVC v142

What is a Type?

0x43	0x50	0x50	0x43	0x4F	0x4E	0x21	0x00

Type	Value*
`int[2]`	`{1129336899, 2182735}`
`double`	`4.81336e-308`
`float[2]`	`{208.314, 3.05866e-39}`
`size_t`	`9374776570171459`
`struct { uint16_t, uint16_t}[2]`	`{{20547, 17232}, {20047, 33}}`

*64-bit MSVC v142

What is a Type?

0x43	0x50	0x50	0x43	0x4F	0x4E	0x21	0x00

Type	Value*
int[2]	{1129336899, 2182735}
double	4.81336e-308
float[2]	{208.314, 3.05866e-39}
size_t	9374776570171459
struct { uint16_t, uint16_t}[2]	{{20547, 17232}, {20047, 33}}
char[8]	“CPPCON!”

*64-bit MSVC v142

Automatic Type Conversions

Automatic Conversions

Allows the compiler to choose a sequence of operations to convert from one type to another without explicitly telling it to do so.

*64-bit MSVC v142

Implicit Type Conversion

Assignment	Value*
float f = -42.1234;	-42.1234016

*64-bit MSVC v142

Implicit Type Conversion

Assignment	Value*
float f = -42.1234;	-42.1234016
double d = f;	-42.123401641845703

*64-bit MSVC v142

Implicit Type Conversion

Assignment	Value*
float f = -42.1234;	-42.1234016
double d = f;	-42.123401641845703
int i = f;	-42

*64-bit MSVC v142

Implicit Type Conversion

Assignment	Value*
float f = -42.1234;	-42.1234016
double d = f;	-42.123401641845703
int i = f;	-42
size_t s = f;	18446744073709551574

*64-bit MSVC v142

Implicit Type Conversion

Assignment	Value*
float f = -42.1234;	-42.1234016
double d = f;	-42.123401641845703
int i = f;	-42
size_t s = f;	18446744073709551574
char c = f;	Ö

*64-bit MSVC v142

Implicit Type Conversion

Assignment	Value*
float f = -42.1234;	-42.1234016
double d = f;	-42.123401641845703
int i = f;	-42
size_t s = f;	18446744073709551574
char c = f;	Ö

struct Float {
  explicit Float(float f_): m{f_}{};
  float m;
};
Float fl = f;
	{ .m = -42.1234}

*64-bit MSVC v142

Arithmetic Type Conversion

```
void drawLine(uint8_t start, uint8_t end);

uint8_t x = 10;
uint8_t width = 50;
drawLine(x, x + width);
```

```
<source>:47:19: warning: conversion from 'int' to 'uint8_t' {aka 'unsigned char'} may change value [-Warith-conversion]
   47 |     drawLine(x, x + width);
      |                 ~~^~~~~~~
```

https://en.cppreference.com/w/cpp/language/operator_arithmetic#Conversions

Sign Conversion

```
struct region { int size; };

void init_region(char* backing_buffer, size_t buffer_size ) {
  if(buffer_size < current_region.size) {
    LOG(“Buffer size too small”);
    return;
  }
//other init code
}
```

```
<source>:13:18: warning: comparison of integer expressions of different
signedness: 'size_t' {aka 'long unsigned int'} and 'int' [-Wsign-compare]
   13 |   if(buffer_size < current_region.size) {
      |      ~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~
```

User conversion operators

```
struct SuperInt {
  operator int() const {
    return mIntRep;
  }
  int mIntRep;
};

SuperInt si;
int i = si;
```

Explicit Type Conversions

Explicit Type Conversions

a.k.a Casts

C-style cast

(<type>)var

1. Create a temporary of <type> using var
2. <type> can be any valid type with qualifiers
3. Overrides the type system by changing the meaning of the bits in a variable
4. Will fail to compile under some circumstances (more later)
5. Can be used in `constexpr` context (more later)
6. Can cause undefined behavior
7. Participates in operator precedence (level 3)

https://en.cppreference.com/w/cpp/language/operator_precedence

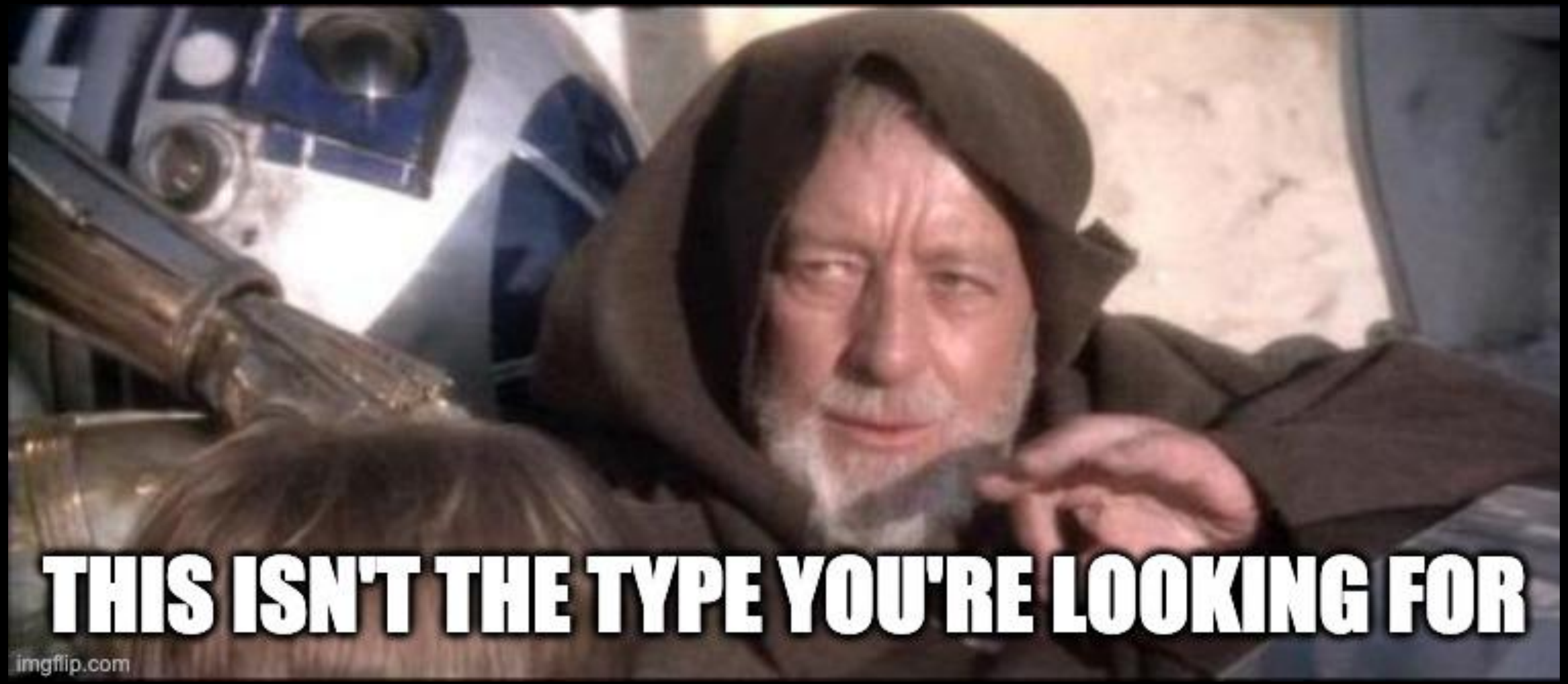
C-style cast

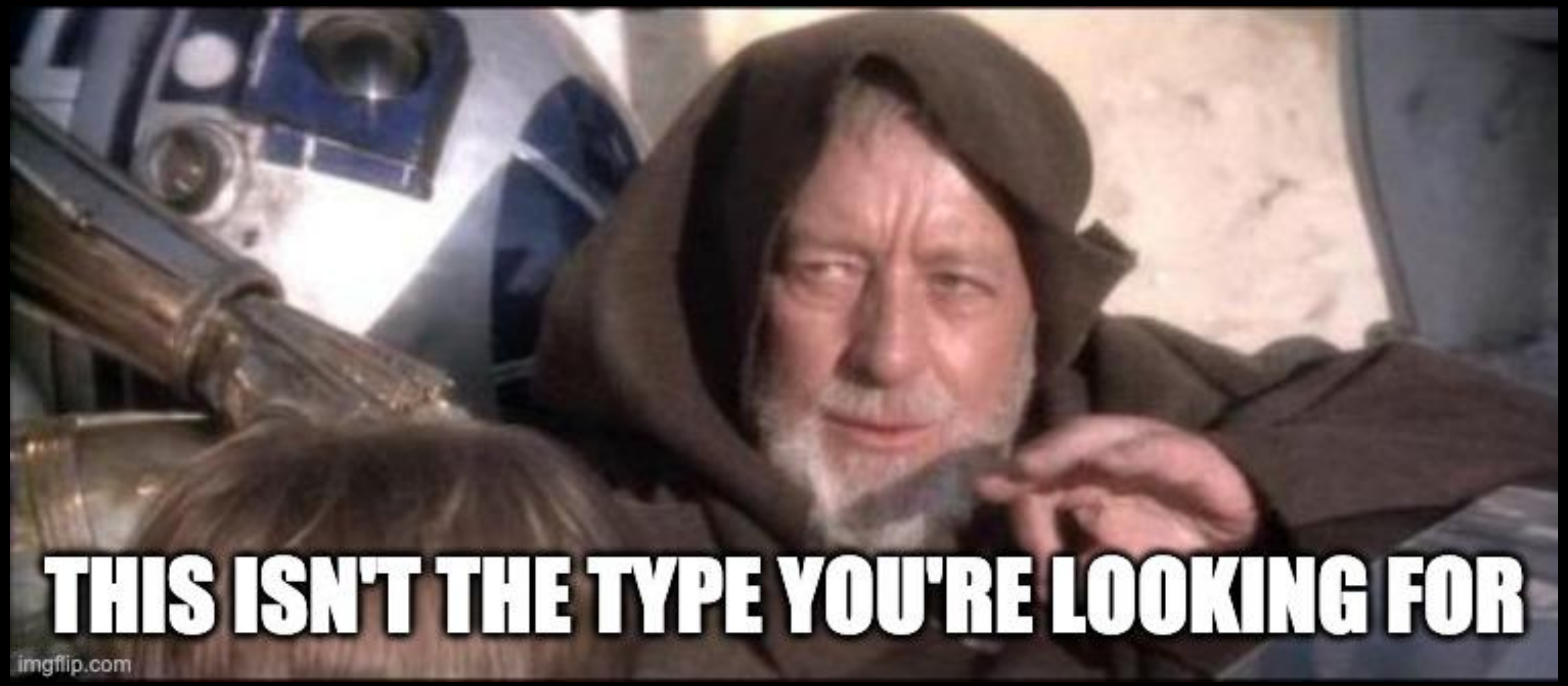
(<type>)var

```
struct A{};
struct B{};

int main() {
 float f = 7.406f;
 int i = (int)f;
 A* pa = (A*)&f;
 B* pb = (B*)pa;
 double d = *(double*)(pb);
 return (int)d;
}
```

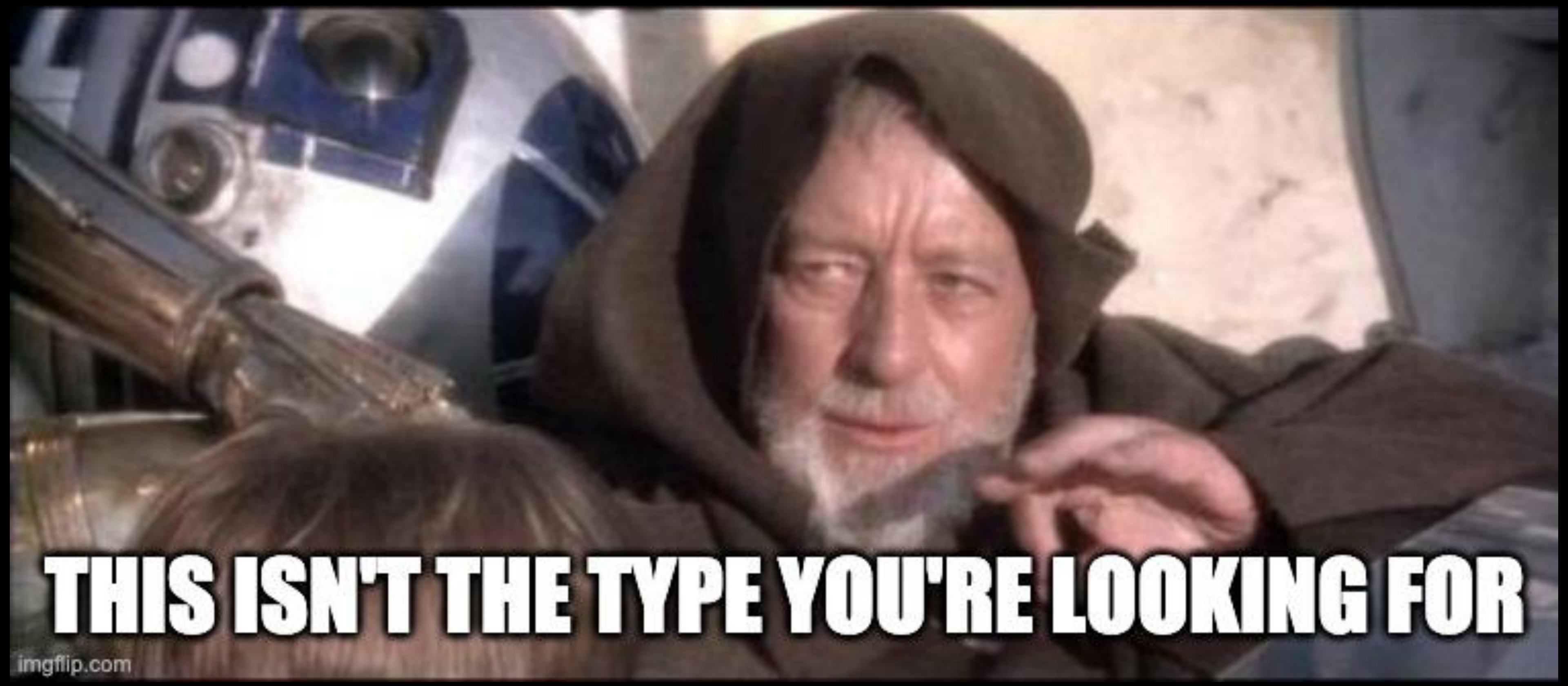
1. Create a temporary of <type> using var
2. <type> can be any valid type with qualifiers
3. Overrides the type system by changing the meaning of the bits in a variable
4. Will fail to compile under some circumstances (more later)
5. Can be used in `constexpr` context (more later)
6. Can cause undefined behavior
7. Participates in operator precedence (level 3)

https://en.cppreference.com/w/cpp/language/operator_precedence

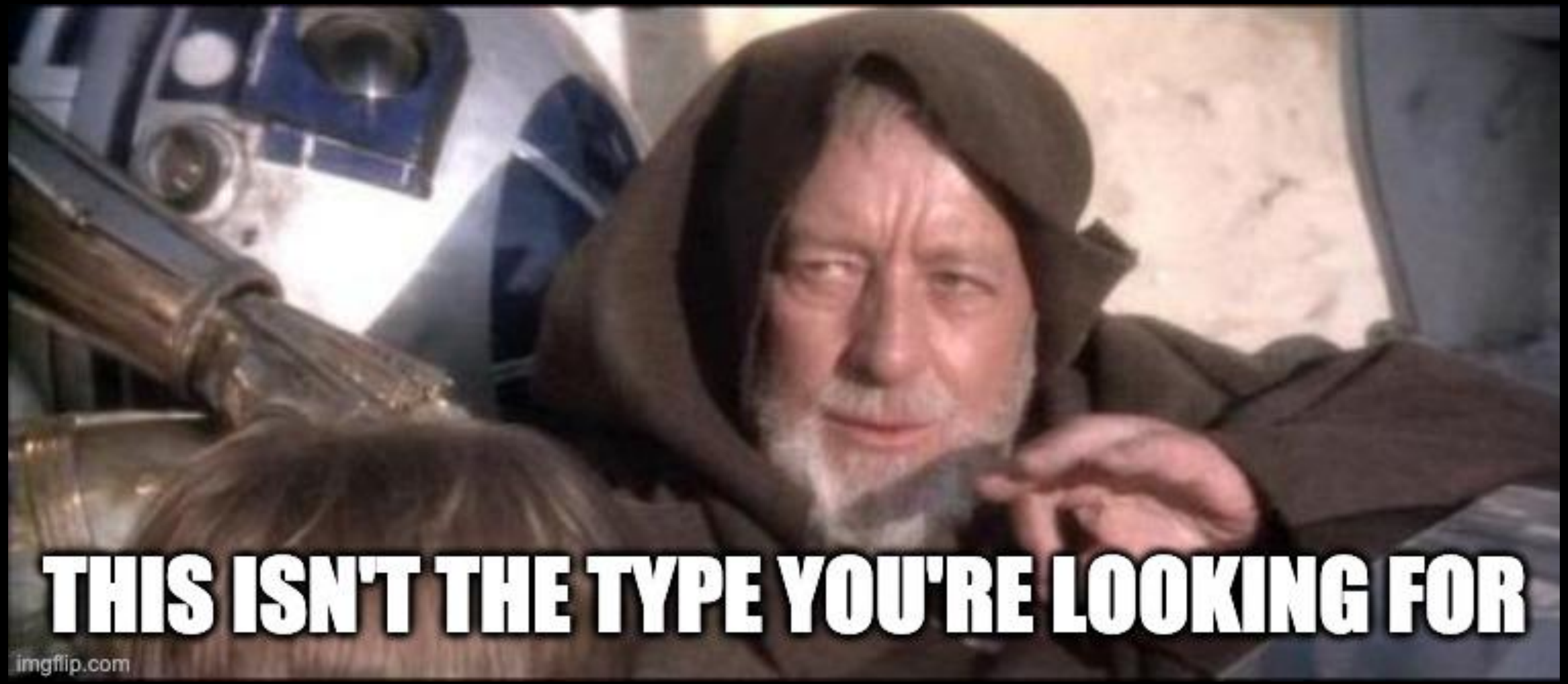

THIS ISN'T THE TYPE YOU'RE LOOKING FOR
imgflip.com



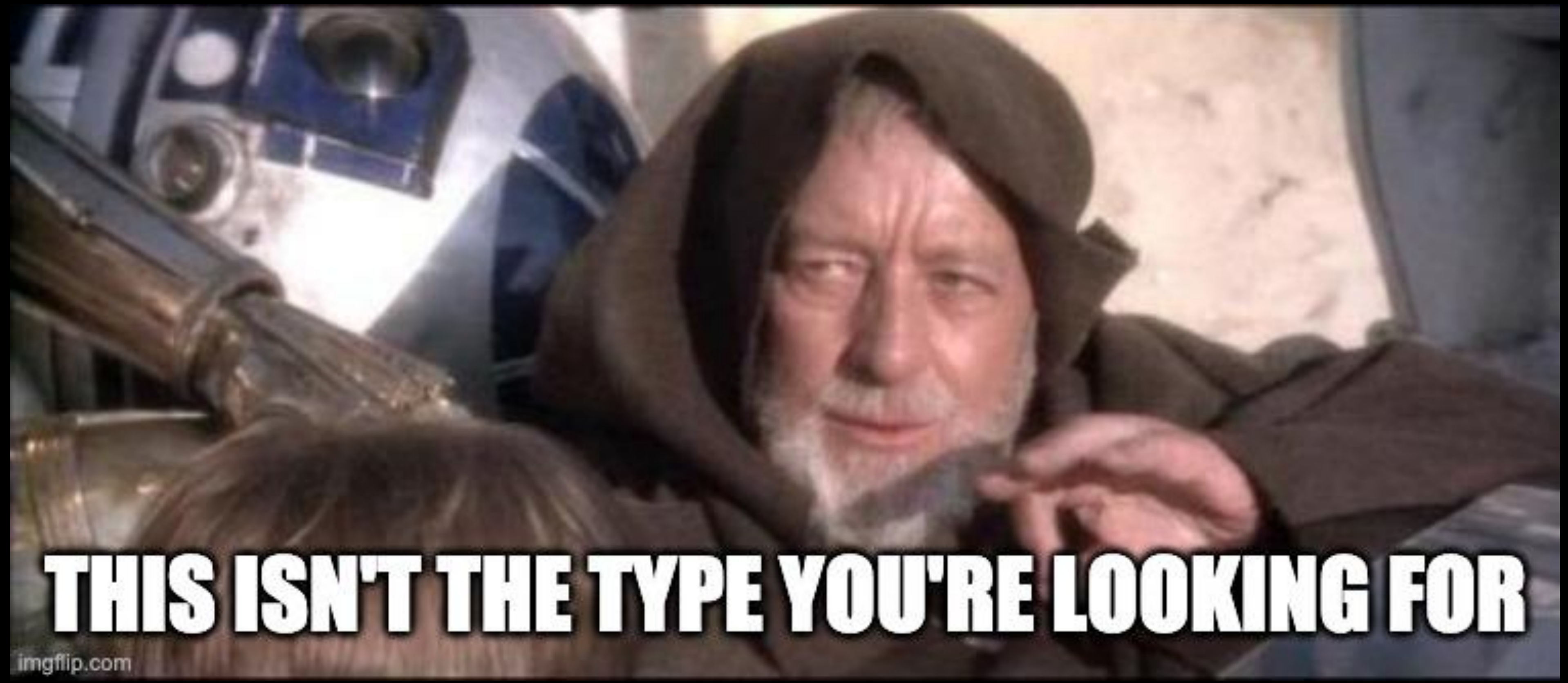

```
struct tree { bool has_leaves = true;};
struct car {int model_year = 1982; };
```




```
struct tree { bool has_leaves = true;};
struct car {int model_year = 1982; };

void prune(tree* t) { t->has_leaves = false; }
void drive(const car* c ) {
 printf("Driving %d\r\n", c->model_year);
}
```

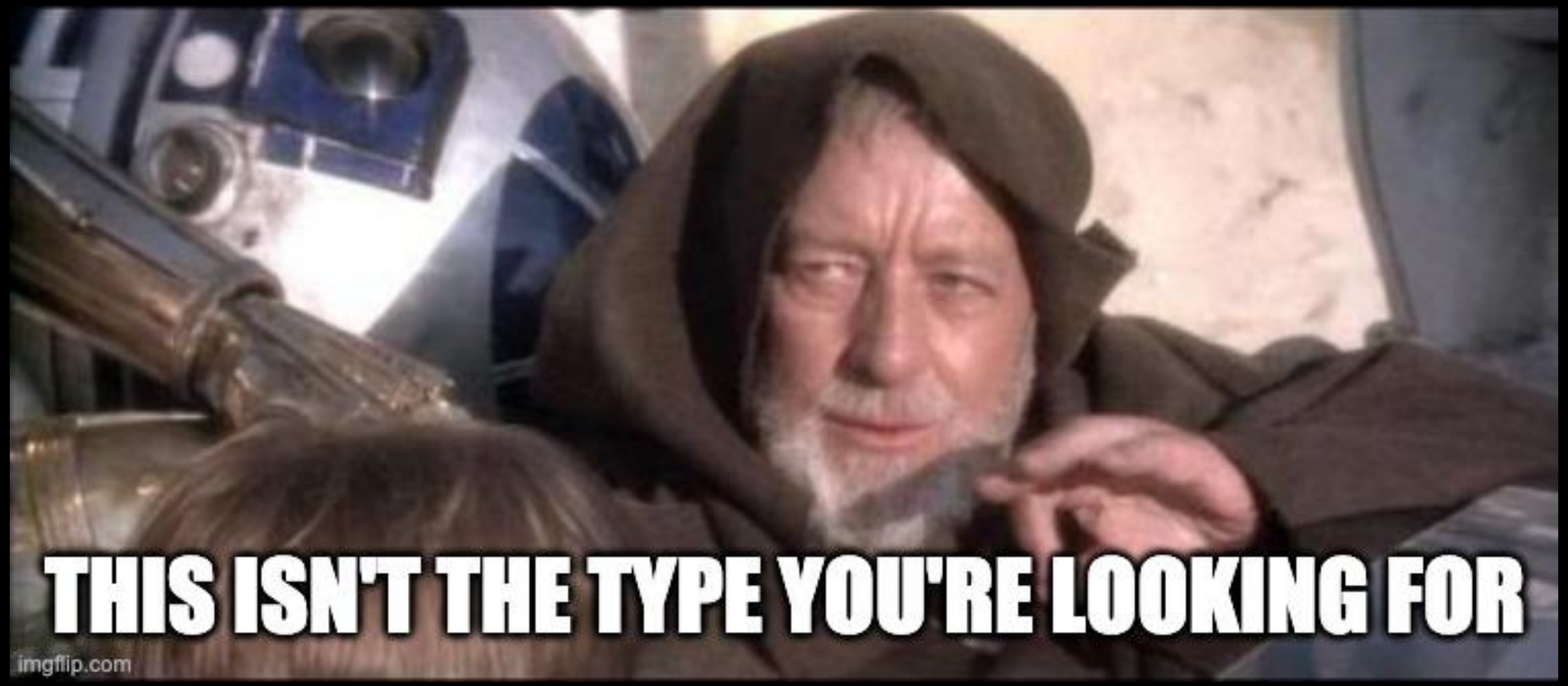



```
struct tree { bool has_leaves = true;};
struct car {int model_year = 1982; };

void prune(tree* t) { t->has_leaves = false; }
void drive(const car* c ) {
 printf("Driving %d\r\n", c->model_year);
}

int main() {
 const tree oak;
 car mustang;
```



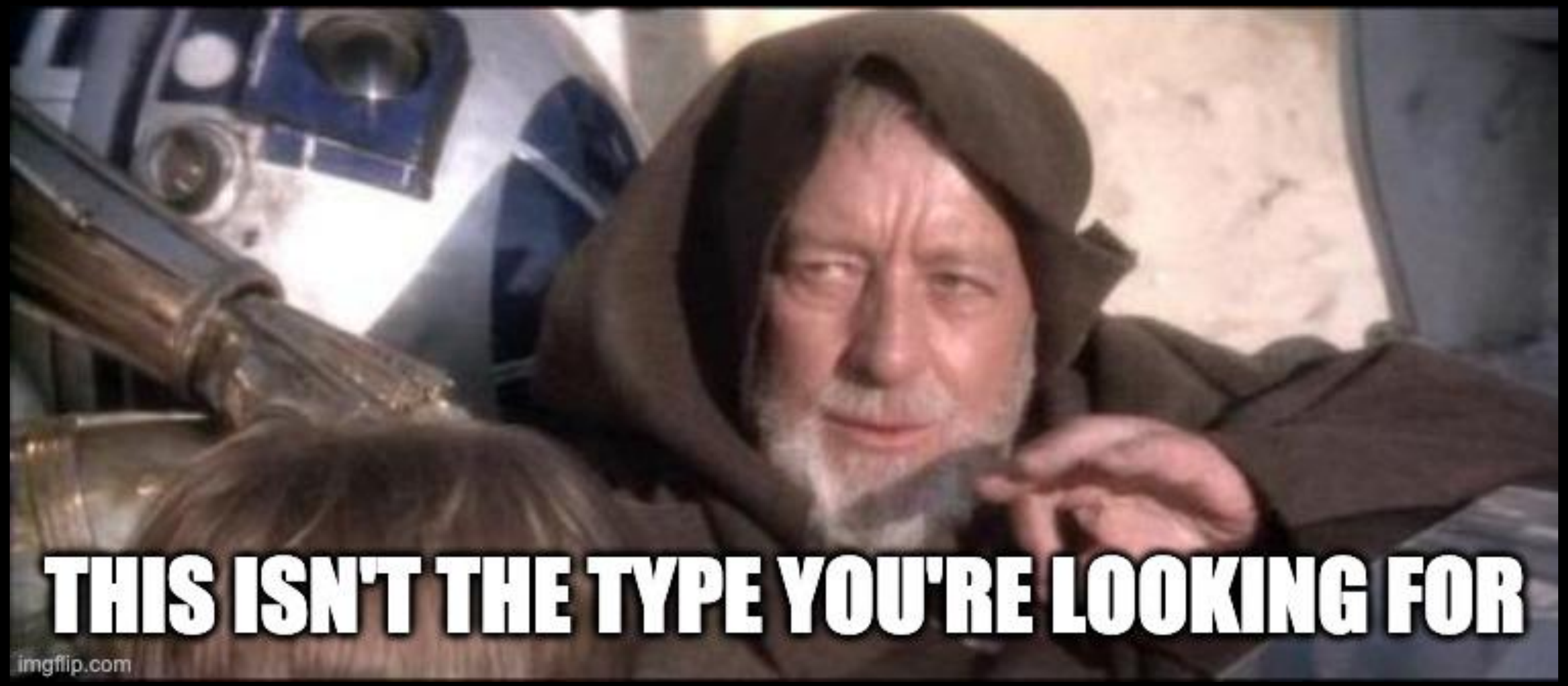

```
struct tree { bool has_leaves = true;};
struct car {int model_year = 1982; };

void prune(tree* t) { t->has_leaves = false; }
void drive(const car* c ) {
 printf("Driving %d\r\n", c->model_year);
}

int main() {
 const tree oak;
 car mustang;

 drive(&mustang); //normal function call
```

```
Driving 1982
```



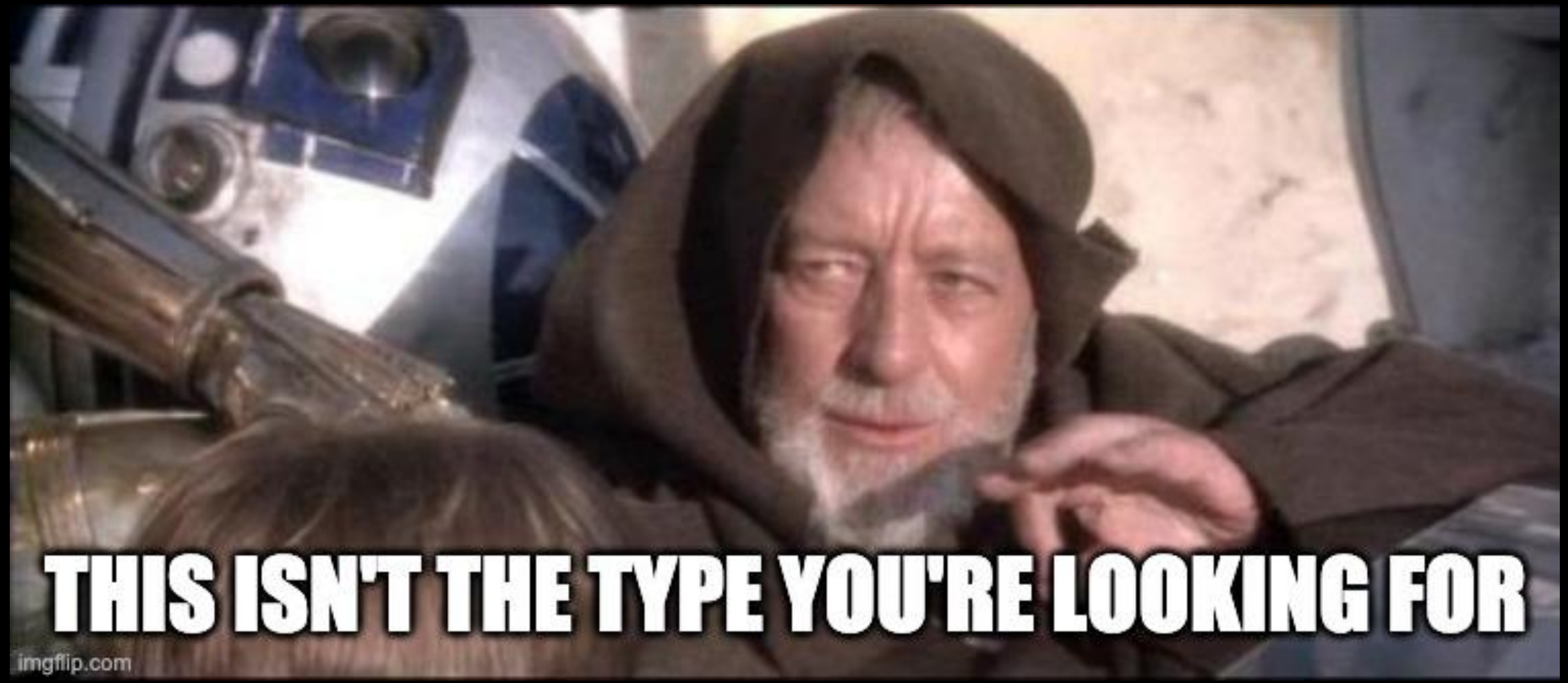

```
struct tree { bool has_leaves = true;};
struct car {int model_year = 1982; };

void prune(tree* t) { t->has_leaves = false; }
void drive(const car* c ) {
 printf("Driving %d\r\n", c->model_year);
}

int main() {
 const tree oak;
 car mustang;

 drive(&mustang); //normal function call
 prune((tree*)&oak); //pruning a const tree
```

```
Driving 1982
```



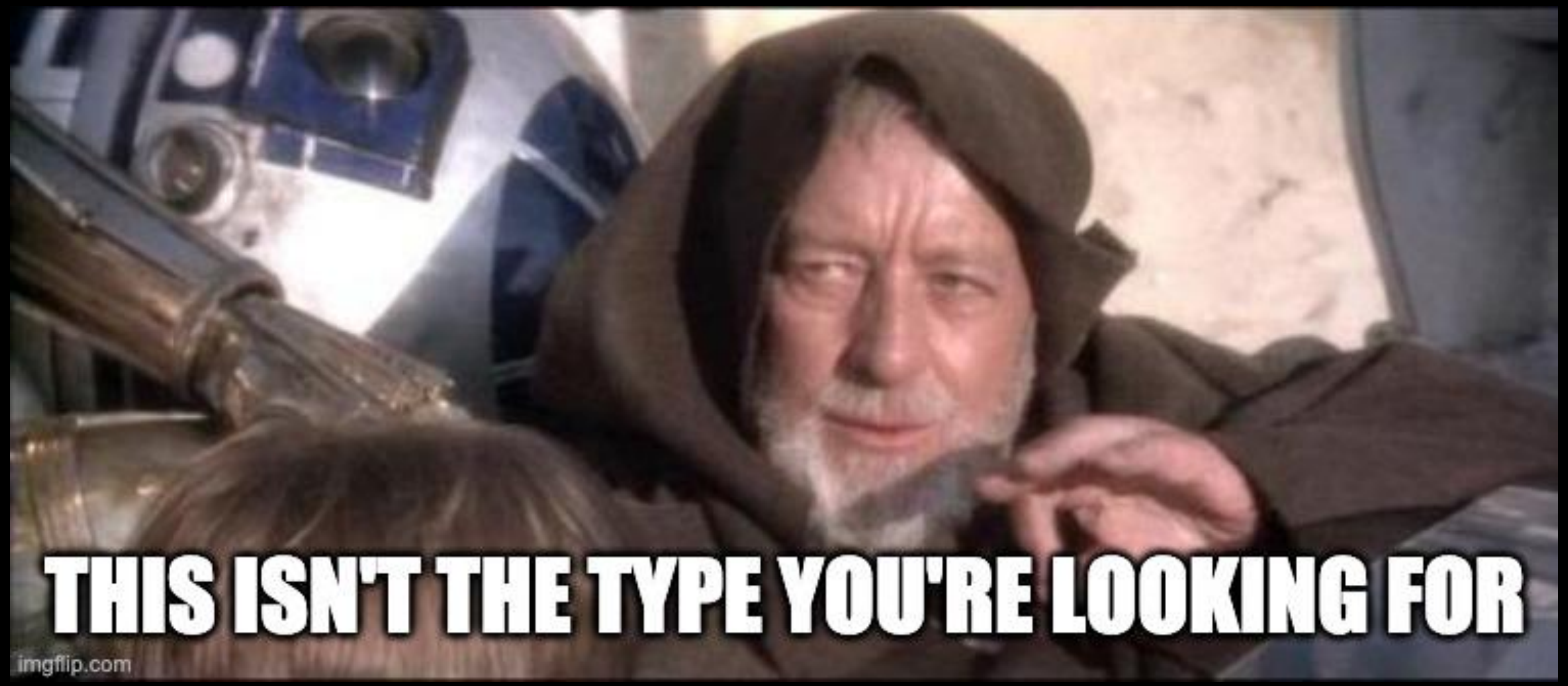

```
struct tree { bool has_leaves = true;};
struct car {int model_year = 1982; };

void prune(tree* t) { t->has_leaves = false; }
void drive(const car* c ) {
 printf("Driving %d\r\n", c->model_year);
}

int main() {
 const tree oak;
 car mustang;

 drive(&mustang); //normal function call
 prune((tree*)&oak); //pruning a const tree
 drive((car*)&oak); // driving a tree
```

```
Driving 1982
Driving 0
```




```
struct tree { bool has_leaves = true;};
struct car {int model_year = 1982; };

void prune(tree* t) { t->has_leaves = false; }
void drive(const car* c ) {
 printf("Driving %d\r\n", c->model_year);
}

int main() {
 const tree oak;
 car mustang;

 drive(&mustang); //normal function call
 prune((tree*)&oak); //pruning a const tree
 drive((car*)&oak); // driving a tree
 prune((tree*)&mustang); // pruning a car
```

```
Driving 1982
Driving 0
```




```
struct tree { bool has_leaves = true;};
struct car {int model_year = 1982; };

void prune(tree* t) { t->has_leaves = false; }
void drive(const car* c ) {
 printf("Driving %d\r\n", c->model_year);
}

int main() {
 const tree oak;
 car mustang;

 drive(&mustang); //normal function call
 prune((tree*)&oak); //pruning a const tree
 drive((car*)&oak); // driving a tree
 prune((tree*)&mustang); // pruning a car
 drive(&mustang); // driving a car from 1792
```

```
Driving 1982
Driving 0
Driving 1792
```

Function pointers are types, so you can cast them

Function pointers are types, so you can cast them

Type	Syntax	Example
function pointer	(<return>(*)(<args...>))<var>	(int(*)(int))f
pointer to member function	(<return> <class>::*(<args...>))<var>	(int (S::*)(int))s_memberptr;

Function pointers are types, so you can cast them

Type	Syntax	Example
function pointer	(<return>(*)(<args...>))<var>	(int(*)(int))f
pointer to member function	(<return> <class>::*(<args...>))<var>	(int (S::*)(int))s_memberptr;

```
void run_function(void* fptr) {
  auto* f = (void(*)(int))fptr;
  f(7);
}

void someFunc(int i) {printf("%d\r\n", i);}

int main( ) {
 run_function((void*)someFunc);
 return 0;
}
```

```
Program returned: 0
  7
```

C++ Functional Cast (a.k.a Constructor Call Notation)

<type>(var)

1. Creates a temporary <type> from var
2. Provides parity with C++ constructors for built in types
3. Can only use a single word type name
4. Participates in operator precedence (level 2)

C++ Functional Cast (a.k.a Constructor Call Notation)

<type>(var)

```
struct A{virtual ~A() = default;};
struct B:public A{};
struct C{};
```

1. Creates a temporary <type> from var
2. Provides parity with C++ constructors for built in types
3. Can only use a single word type name
4. Participates in operator precedence (level 2)

C++ Functional Cast (a.k.a Constructor Call Notation)

<type>(var)

```
struct A{virtual ~A() = default;};
struct B:public A{};
struct C{};

template <typename T, typename F>
T convert_to(F& f) { return T(f); }
```

1. Creates a temporary <type> from var
2. Provides parity with C++ constructors for built in types
3. Can only use a single word type name
4. Participates in operator precedence (level 2)

C++ Functional Cast (a.k.a Constructor Call Notation)

<type>(var)

```
struct A{virtual ~A() = default;};
struct B:public A{};
struct C{};

template <typename T, typename F>
T convert_to(F& f) { return T(f); }

int main() {
 int i = 7;
 float f = convert_to<float>(i);
```

1. Creates a temporary <type> from var
2. Provides parity with C++ constructors for built in types
3. Can only use a single word type name
4. Participates in operator precedence (level 2)

C++ Functional Cast (a.k.a Constructor Call Notation)

<type>(var)

```
struct A{virtual ~A() = default;};
struct B:public A{};
struct C{};

template <typename T, typename F>
T convert_to(F& f) { return T(f); }

int main() {
 int i = 7;
 float f = convert_to<float>(i);
 //A* pa = A*(&f); //<-- Will not compile
```

1. Creates a temporary <type> from var
2. Provides parity with C++ constructors for built in types
3. Can only use a single word type name
4. Participates in operator precedence (level 2)

C++ Functional Cast (a.k.a Constructor Call Notation)

<type>(var)

```
struct A{virtual ~A() = default;};
struct B:public A{};
struct C{};

template <typename T, typename F>
T convert_to(F& f) { return T(f); }

int main() {
 int i = 7;
 float f = convert_to<float>(i);
 //A* pa = A*(&f); //<-- Will not compile
 using astar = A*;
 A* pa = astar(&f);
 C* pc = convert_to<C*>(pa);
```

1. Creates a temporary <type> from var
2. Provides parity with C++ constructors for built in types
3. Can only use a single word type name
4. Participates in operator precedence (level 2)

C++ Functional Cast (a.k.a Constructor Call Notation)

<type>(var)

```
struct A{virtual ~A() = default;};
struct B:public A{};
struct C{};

template <typename T, typename F>
T convert_to(F& f) { return T(f); }

int main() {
 int i = 7;
 float f = convert_to<float>(i);
 //A* pa = A*(&f); //<-- Will not compile
 using astar = A*;
 A* pa = astar(&f);
 C* pc = convert_to<C*>(pa);
 B& rb = convert_to<B&>(*pa);
 return 0;
}
```

1. Creates a temporary <type> from var
2. Provides parity with C++ constructors for built in types
3. Can only use a single word type name
4. Participates in operator precedence (level 2)

Problems with C-style and Functional notation casts

1. Single notation, multiple meanings
2. Error prone
3. Not grep-able
4. Complicate the C and C++ grammar

Goals for C++ casting

1. Different notation or different tasks
2. Easily recognized and searchable
3. Perform all operations that C casts can
4. Eliminate unintended errors

Goals for C++ casting

1. Different notation or different tasks
2. Easily recognized and searchable
3. Perform all operations that C casts can
4. Eliminate unintended errors
5. Make casting less enticing

C++ casting operators*

1. `static_cast`
2. `const_cast`
3. `dynamic_cast`
4. `reinterpret_cast`

*keywords

static_cast<T>

```
struct B {};
struct D : public B {};

int main() {
 int i = 7001;
 float f = static_cast<float>(i); // 1
 uint8_t ui8 = static_cast<uint8_t>(1.75f * f); // 2
 D d;
 B& rb = d;
 D& rd = static_cast<D&>(rb); //3
 return 0;
}

void* some_c_handler(void* pv) {
  B* pb = static_cast<B*>(pv); // 4
  return pb;
}
```

1. Creates a temporary of type from var
2. Tries to find a path from T1 to T2 via implicit and user-defined conversion or construction. Cannot *remove* CV qualification.
3. Use when you want to:
   1. Clarify implicit conversion
   2. Indicate intentional truncation
   3. Cast between base and derived
   4. Cast between `void*` and `T*`

GARMIN.

`static_cast<T>` multiple hops

```
struct A { explicit A(int){ puts("A");}};
```

1. A has a constructor that takes a single `int`

`static_cast<T>` multiple hops

```
struct A { explicit A(int){ puts("A");}};
struct E {
  operator int(){
    puts("B::operator int");
    return 0;
  }
};
```

1. A has a constructor that takes a single `int`
2. E has a user defined conversion to `int`

static_cast<T> multiple hops

```
struct A { explicit A(int){ puts("A");}};
struct E {
  operator int(){
    puts("B::operator int");
    return 0;
  }
};

int main() {
 E e;
 A a = static_cast<A>(e);
 return 0;
}
```

1. A has a constructor that takes a single `int`
2. E has a user defined conversion to `int`

static_cast<T> multiple hops

```
struct A { explicit A(int){ puts("A");}};
struct E {
  operator int(){
    puts("B::operator int");
    return 0;
  }
};

int main() {
 E e;
 A a = static_cast<A>(e);
 return 0;
}
```

1. A has a constructor that takes a single `int`
2. E has a user defined conversion to `int`

static_cast<T> multiple hops

```
struct A { explicit A(int){ puts("A");}};
struct E {
  operator int(){
    puts("B::operator int");
    return 0;
  }
};

int main() {
 E e;
 A a = static_cast<A>(e);
 return 0;
}
```

1. A has a constructor that takes a single `int`
2. E has a user defined conversion to `int`

`static_cast<T>` multiple hops

```
struct A { explicit A(int){ puts("A");}};
struct E {
  operator int(){
    puts("B::operator int");
    return 0;
  }
};

int main() {
 E e;
 A a{static_cast<int>(e)};
 return 0;
}
```

1. A has a constructor that takes a single `int`
2. E has a user defined conversion to `int`

static_cast<T> and inheritance

```
struct Base1 { virtual ~Base1() = default; int i;};
struct Base2 { virtual ~Base2() = default; int j;};
struct Derived : public Base1, public Base2 { int k; };
```

static_cast<T> and inheritance

```
struct Base1 { virtual ~Base1() = default; int i;};
struct Base2 { virtual ~Base2() = default; int j;};
struct Derived : public Base1, public Base2 { int k; };
void CheckSame(void* p1, void* p2) {
    if (p1 == p2) { puts("Same!");}
    else { puts("Different!"); }
}
```

static_cast<T> and inheritance

```
struct Base1 { virtual ~Base1() = default; int i;};
struct Base2 { virtual ~Base2() = default; int j;};
struct Derived : public Base1, public Base2 { int k; };
void CheckSame(void* p1, void* p2) {
    if (p1 == p2) { puts("Same!");}
    else { puts("Different!"); }
}

int main() {
    Derived d;
    Derived* derivedptr = &d;
    Base1* base1ptr = static_cast<Base1*>(&d);
    CheckSame(derivedptr, base1ptr);
```

```
Program returned: 0
    Same!
```

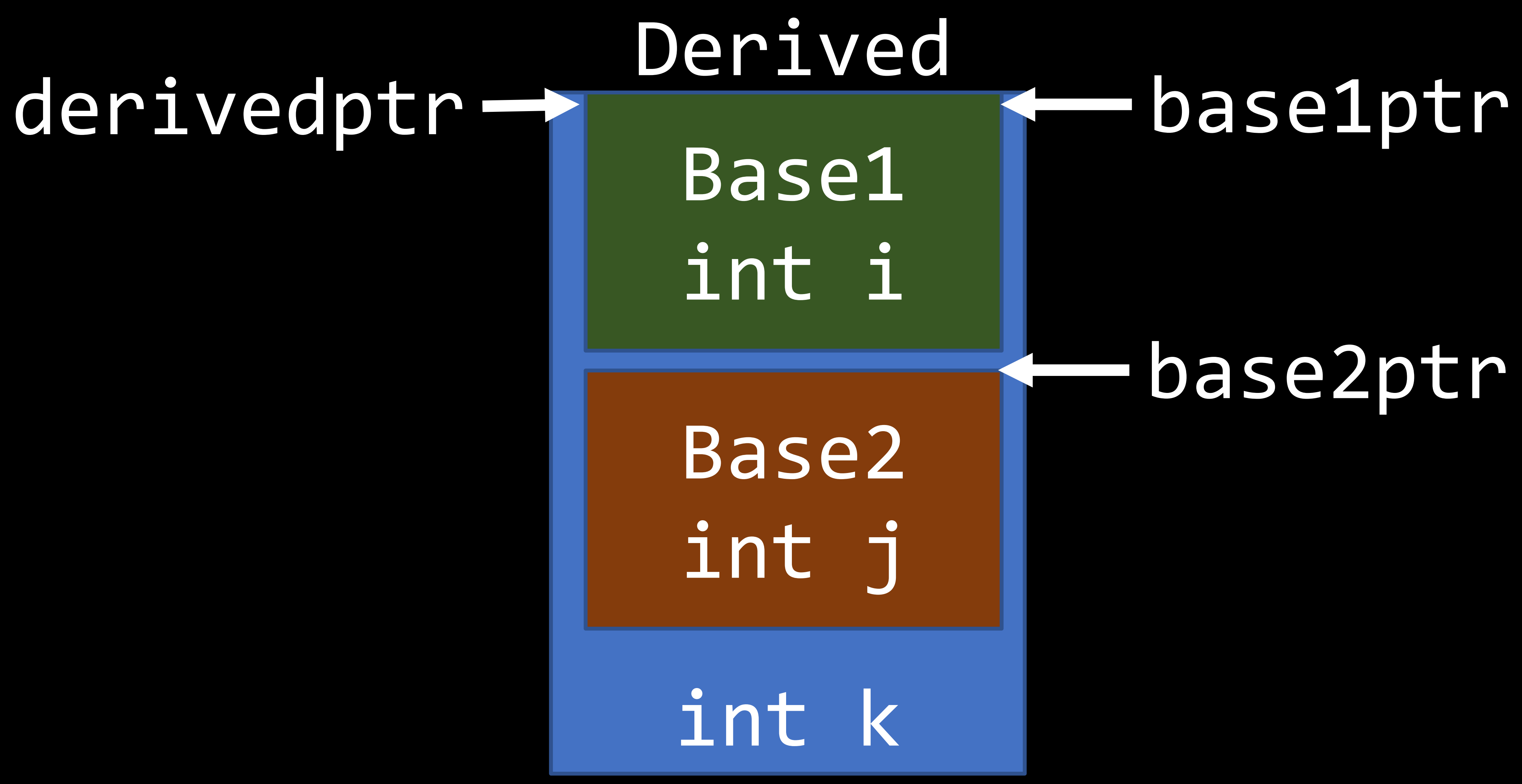
static_cast<T> and inheritance

```
struct Base1 { virtual ~Base1() = default; int i;};
struct Base2 { virtual ~Base2() = default; int j;};
struct Derived : public Base1, public Base2 { int k; };
void CheckSame(void* p1, void* p2) {
    if (p1 == p2) { puts("Same!");}
    else { puts("Different!"); }
}

int main() {
    Derived d;
    Derived* derivedptr = &d;
    Base1* base1ptr = static_cast<Base1*>(&d);
    CheckSame(derivedptr, base1ptr);

    Base2* base2ptr = static_cast<Base2*>(&d);
    CheckSame(derivedptr, base2ptr);

    return 0;
}
```

```
Program returned: 0
    Same!
    Different!
```

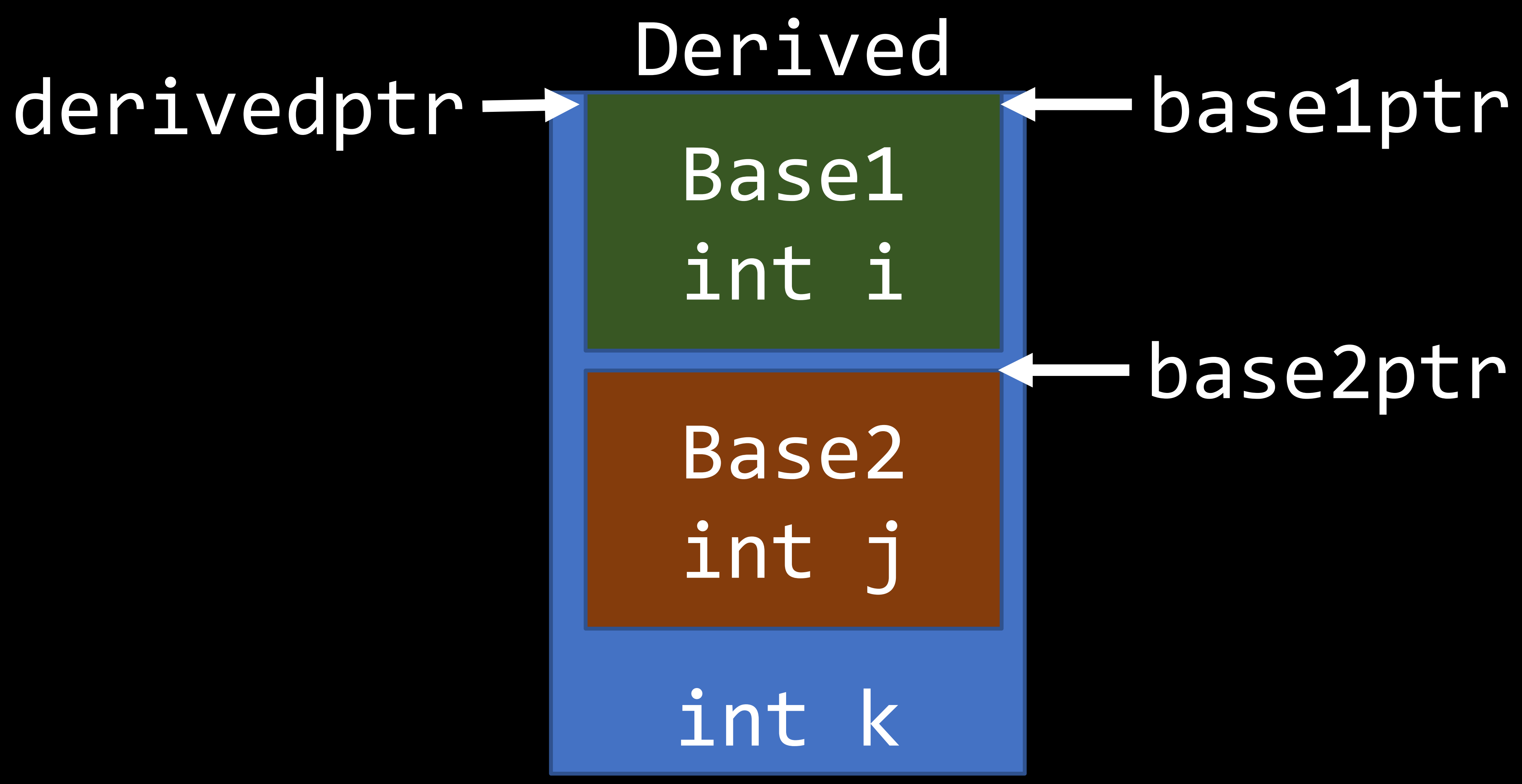
static_cast<T> and inheritance

- `static_cast` to one of the base types will offset the pointer into the derived type to the base type's location



static_cast<T> and inheritance

```cpp
struct Base1 { virtual ~Base1() = default; int i;};
struct Base2 { virtual ~Base2() = default; int j;};
struct Derived : public Base1, public Base2 { int k; };
void CheckSame(void* p1, void* p2) {
    if (p1 == p2) { puts("Same!");}
    else { puts("Different!"); }
}

int main() {
    Derived d;
    Derived* derivedptr = &d;
    Base1* base1ptr = static_cast<Base1*>(&d);
    CheckSame(derivedptr, base1ptr);

    Base2* base2ptr = static_cast<Base2*>(&d);
    CheckSame(derivedptr, base2ptr);

    void* derived_plus_offset =
        (char*)derivedptr + sizeof(Base1);
    CheckSame(derived_plus_offset, base2ptr);
    return 0;
}
```




```
Program returned: 0
    Same!
    Different!
    Same!
```

`static_cast<T>` is not infallible

```
struct Base { virtual ~Base() = default;
  virtual void f() { puts("base");}
};

struct Derived : public Base {
  void f() override { puts("Derived");}
};

struct other : public Base {
    void f() override { puts("other");}
};
```

- `static_cast` does not protect against downcasting to an unrelated type

static_cast<T> is not infallible

```
struct Base { virtual ~Base() = default;
  virtual void f() { puts("base");}
};

struct Derived : public Base {
  void f() override { puts("Derived");}
};

struct other : public Base {
    void f() override { puts("other");}
};

int main() {
  Derived d;
  Base& b = d;
  d.f();
  b.f();
```

- `static_cast` does not protect against downcasting to an unrelated type

```
Program returned: 0
  Derived
  Derived
```

`static_cast<T>` is not infallible

```
struct Base { virtual ~Base() = default;
  virtual void f() { puts("base");}
};

struct Derived : public Base {
  void f() override { puts("Derived");}
};

struct other : public Base {
    void f() override { puts("other");}
};

int main() {
  Derived d;
  Base& b = d;
  d.f();
  b.f();
  other& a = static_cast<other&>(b);
  a.f();
```

- `static_cast` does not protect against downcasting to an unrelated type

```
Program returned: 0
  Derived
  Derived
  Derived
```

static_cast<T> is not infallible

```
struct Base { virtual ~Base() = default;
  virtual void f() { puts("base");}
};

struct Derived : public Base {
  void f() override { puts("Derived");}
};

struct other : public Base {
    void f() override { puts("other");}
};

int main() {
  Derived d;
  Base& b = d;
  d.f();
  b.f();
  other& a = static_cast<other&>(b);
  a.f();
  static_assert(std::is_same<decltype(a), other&>::value,
    "not the same");
  return 0;
}
```

- `static_cast` does not protect against downcasting to an unrelated type

```
Program returned: 0
  Derived
  Derived
  Derived
```

`const_cast<T>`

1. Removes or adds `const` or `volatile` qualifiers from or to a variable, cannot change type
2. Does NOT change the CV qualification of the original variable

const_cast<T>

```
void use_pointer( int* pi) {
 printf("Use %d\r\n", *pi);
}
void modify_pointer( int* pi) { *pi = 42; }
```

1. Removes or adds `const` or `volatile` qualifiers from or to a variable, cannot change type
2. Does NOT change the CV qualification of the original variable

const_cast<T>

```
void use_pointer( int* pi) {
 printf("Use %d\r\n", *pi);
}
void modify_pointer( int* pi) { *pi = 42; }

int main() {
 const int i = 7;
```

1. Removes or adds `const` or `volatile` qualifiers from or to a variable, cannot change type
2. Does NOT change the CV qualification of the original variable

const_cast<T>

```
void use_pointer( int* pi) {
 printf("Use %d\r\n", *pi);
}
void modify_pointer( int* pi) { *pi = 42; }

int main() {
 const int i = 7;

 use_pointer(const_cast<int*>(&i));
```

1. Removes or adds `const` or `volatile` qualifiers from or to a variable, cannot change type
2. Does NOT change the CV qualification of the original variable

```
Program returned: 0
    Use 7
```

const_cast<T>

```
void use_pointer( int* pi) {
 printf("Use %d\r\n", *pi);
}
void modify_pointer( int* pi) { *pi = 42; }

int main() {
 const int i = 7;

 use_pointer(const_cast<int*>(&i));
 modify_pointer(const_cast<int*>(&i));
 printf("Modified %d\r\n", i);
```

1. Removes or adds `const` or `volatile` qualifiers from or to a variable, cannot change type
2. Does NOT change the CV qualification of the original variable

```
Program returned: 0
    Use 7
    Modified 7
```

const_cast<T>

```
void use_pointer( int* pi) {
 printf("Use %d\r\n", *pi);
}
void modify_pointer( int* pi) { *pi = 42; }

int main() {
 const int i = 7;

 use_pointer(const_cast<int*>(&i));
 modify_pointer(const_cast<int*>(&i));
 printf("Modified %d\r\n", i);

 int j = 4;
 const int* cj = &j;
 modify_pointer(const_cast<int*>(cj));
 printf("Modified %d\r\n", i);
 return 0;
}
```

1. Removes or adds `const` or `volatile` qualifiers from or to a variable, cannot change type
2. Does NOT change the CV qualification of the original variable

```
Program returned: 0
    Use 7
    Modified 7
    Modified 42
```

const_cast<T> example: member overload

```
struct my_array {
    const char& operator[] (size_t offset) const {
        return buffer[offset];
    }

  private:
    char buffer[10];
};

int main() {
  const my_array a;
  const auto& c = a[4];
  return 0;
}
```

- Used to prevent code duplication for member functions

const_cast<T> example: member overload

```
struct my_array {
    const char& operator[] (size_t offset) const {
        return buffer[offset];
    }

  private:
    char buffer[10];
};

int main() {
  const my_array a;
  const auto& c = a[4];
  my_array mod_a;
  mod_a[4] = 7;

  return 0;
}
```

- Used to prevent code duplication for member functions

const_cast<T> example: member overload

```
struct my_array {
    const char& operator[] (size_t offset) const {
        return buffer[offset];
    }

    char& operator[](size_t offset) {
      return buffer[offset];
    }

  private:
    char buffer[10];
};

int main() {
  const my_array a;
  const auto& c = a[4];
  my_array mod_a;
  mod_a[4] = 7;

  return 0;
}
```

- Used to prevent code duplication for member functions

const_cast<T> example: member overload

```
class my_array {
  public:
    char& operator[](size_t offset) {
      return const_cast<char&>(
        const_cast<const my_array&>(*this)[offset]);
    }
    const char& operator[] (size_t offset) const {
        return buffer[offset];
    }
  private:
    char buffer[10];
};

int main() {
  const my_array a;
  const auto& c = a[4];
  my_array mod_a;
  mod_a[4] = 7;
  return 0;
}
```

- Used to prevent code duplication for member functions

Run Time Type Information (RTTI)

1. Extra information stored for each polymorphic type in an implementation defined struct
2. Allows for querying type information at run time
3. Can be disabled to save space (gcc/clang –fno-rtti, msvc /GR-)

dynamic_cast<T>

```
struct A { virtual ~A() = default; };
struct B : public A { };
struct C : public A {};

int main() {
 C c;
 B b;
 std::vector<A*> a_list = { &c, &b};

 for(size_t i = 0; i < a_list.size(); ++i) {
    A* pa = a_list[i];
    if( dynamic_cast<B*>(pa)){
        printf("a_list[%lu] was a B\r\n", i);
    }
    if( dynamic_cast<C*>(pa)) {
        printf("a_list[%lu] was a C\r\n", i);
    }
 }
 return 0;
}
```

1. See if To is in the same public inheritance tree as From
2. Can only be a reference or pointer
3. Cannot remove CV
4. From must be polymorphic
5. Requires RTTI
6. Returns `nullptr` for pointers and throws `std::bad_cast` for references if the types are not related

```
a_list[0] was a C
a_list[1] was a B
```

`dynamic_cast<T>` example: UI Framework

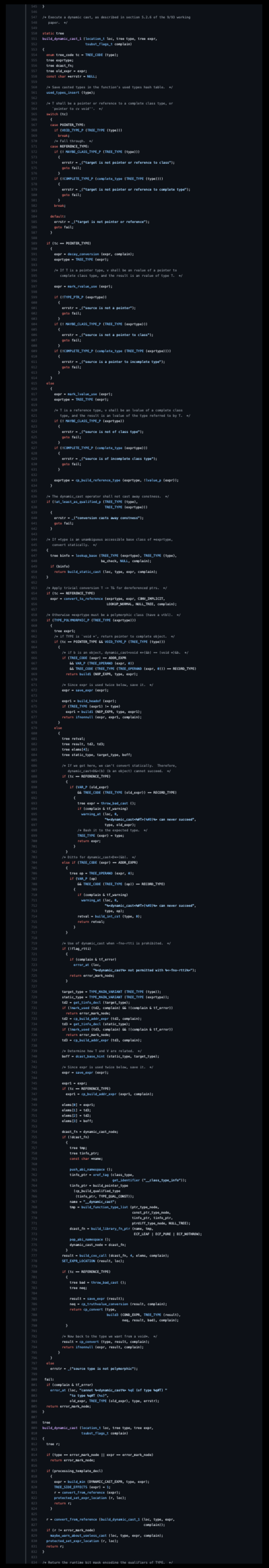
```
struct Widget {};
struct Label : public Widget {};
struct Button : public Widget { void DoClick(); };
```

`dynamic_cast<T>` example: UI Framework

```
struct Widget {};
struct Label : public Widget {};
struct Button : public Widget { void DoClick(); };

struct Page {
std::vector<Widget> mWidgetList;

template<typename T> T* getWidget(WidgetId id) {
 return dynamic_cast<T*>(&mWidgetList[id]);
}
```

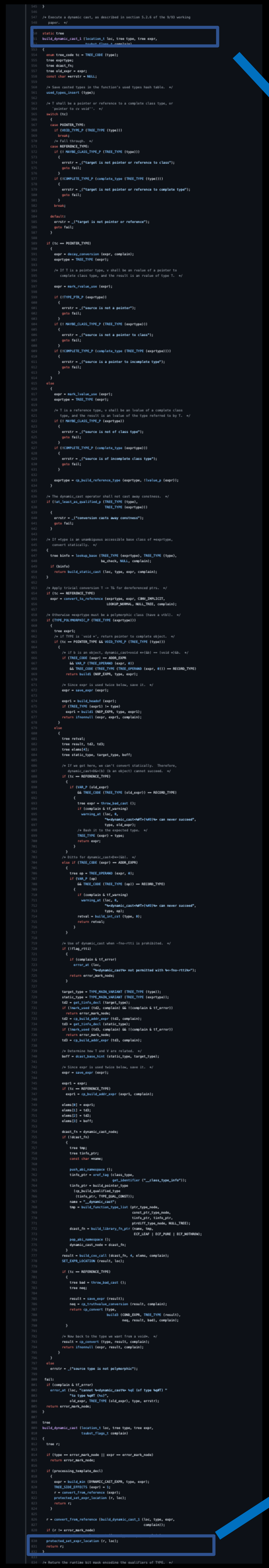
dynamic_cast<T> example: UI Framework

```
struct Widget {};
struct Label : public Widget {};
struct Button : public Widget { void DoClick(); };

struct Page {
std::vector<Widget> mWidgetList;

template<typename T> T* getWidget(WidgetId id) {
 return dynamic_cast<T*>(&mWidgetList[id]);
}

void Page::OnTouch(WidgetId id) {
 auto* touchedWidget = getWidget<Button>(id);
 if(touchedWidget) {
  touchedWidget->DoClick();
 }
 //more processing
}
```



dynamic_cast can be expensive

from gcc’s rtti.c



dynamic_cast can be expensive

from gcc's rtti.c

```
static tree
build_dynamic_cast_1 (location_t loc, tree type, tree expr,
                      tsubst_flags_t complain)
```

```
  return r;
}
```

reinterpret_cast<T>

```
struct A{};
struct B{ int i; int j;};
int main()
{
 int i = 0;
 int* pi = &i;
 uintptr_t uipt = reinterpret_cast<uintptr_t>(pi);
 float& f = reinterpret_cast<float&>(i);
 A a;
 B* pb = reinterpret_cast<B*>(&a);
 char buff[10];
 B* b_buff = reinterpret_cast<B*>(buff);
 return 0;
}
```

1. Can change any pointer or reference type to any other pointer or reference type
2. Also called type-punning
3. Cannot be used in a `constexpr` context
4. Can NOT remove CV qualification
5. Does not ensure sizes of To and From are the same

reinterpret_cast<T>

```
struct A{};
struct B{ int i; int j;};
int main()
{
 int i = 0;
 int* pi = &i;
 uintptr_t uipt = reinterpret_cast<uintptr_t>(pi);
 float& f = reinterpret_cast<float&>(i);
 A a;
 B* pb = reinterpret_cast<B*>(&a);
 char buff[10];
 B* b_buff = reinterpret_cast<B*>(buff);
 volatile int& REGISTER =
   *reinterpret_cast<int*>(0x1234); //6
 return 0;
}
```

1. Can change any pointer or reference type to any other pointer or reference type
2. Also called type-punning
3. Cannot be used in a `constexpr` context
4. Can NOT remove CV qualification
5. Does not ensure sizes of To and From are the same
6. **Useful for memory mapped functionality**

reinterpret_cast<T> accessing private base

```
struct B { void m(){ puts("private to D");}};
struct D: private B {};

int main() {
  D d;
  B& b = reinterpret_cast<B&>(d);
  b.m();
  return 0;
}
```

```
Program returned: 0
     private to D
```

Type Aliasing

The act of using the memory of one type as if it were a different type when the memory layouts of the two types are compatible

https://tinyurl.com/32b3cdjp

Type Aliasing

The act of using the memory of one type as if it were a different type when the memory layouts of the two types are compatible

compatible types

```
struct Point {
 int x;
 int y;
};
struct Location {
 int x;
 int y;
};

Point p{1,2};
auto* loc =
  reinterpret_cast<Location*>(&p);
```

https://tinyurl.com/32b3cdjp

Type Aliasing

The act of using the memory of one type as if it were a different type when the memory layouts of the two types are compatible

compatible types

```
struct Point {
 int x;
 int y;
};
struct Location {
 int x;
 int y;
};

Point p{1,2};
auto* loc =
  reinterpret_cast<Location*>(&p);
```

incompatible types

```
float f = 1.0f;
int* i =
 reinterpret_cast<int*>(&f);
```

https://tinyurl.com/32b3cdjp

Type Aliasing

The act of using the memory of one type as if it were a different type when the memory layouts of the two types are compatible

compatible types?

```
struct Point {
 int x;
 int y;
};
struct Point3D {
 int x;
 int y;
 int z;
};
```

https://tinyurl.com/32b3cdjp

Type Aliasing

The act of using the memory of one type as if it were a different type when the memory layouts of the two types are compatible

compatible types?

```
struct Point {
 int x;
 int y;
};
struct Point3D {
 int x;
 int y;
 int z;
};
```

```
Program returned: 0
  px: 1 p3dx: 1
```

it depends...

```
Point p{1,2};
auto* p3d =
  reinterpret_cast<Point3D*>(&p);
printf("px: %d p3dx: %d", p.x, p3d->x);
```

https://tinyurl.com/32b3cdjp

Type Aliasing

The act of using the memory of one type as if it were a different type when the memory layouts of the two types are compatible

compatible types?

```
struct Point {
 int x;
 int y;
};
struct Point3D {
 int x;
 int y;
 int z;
};
```

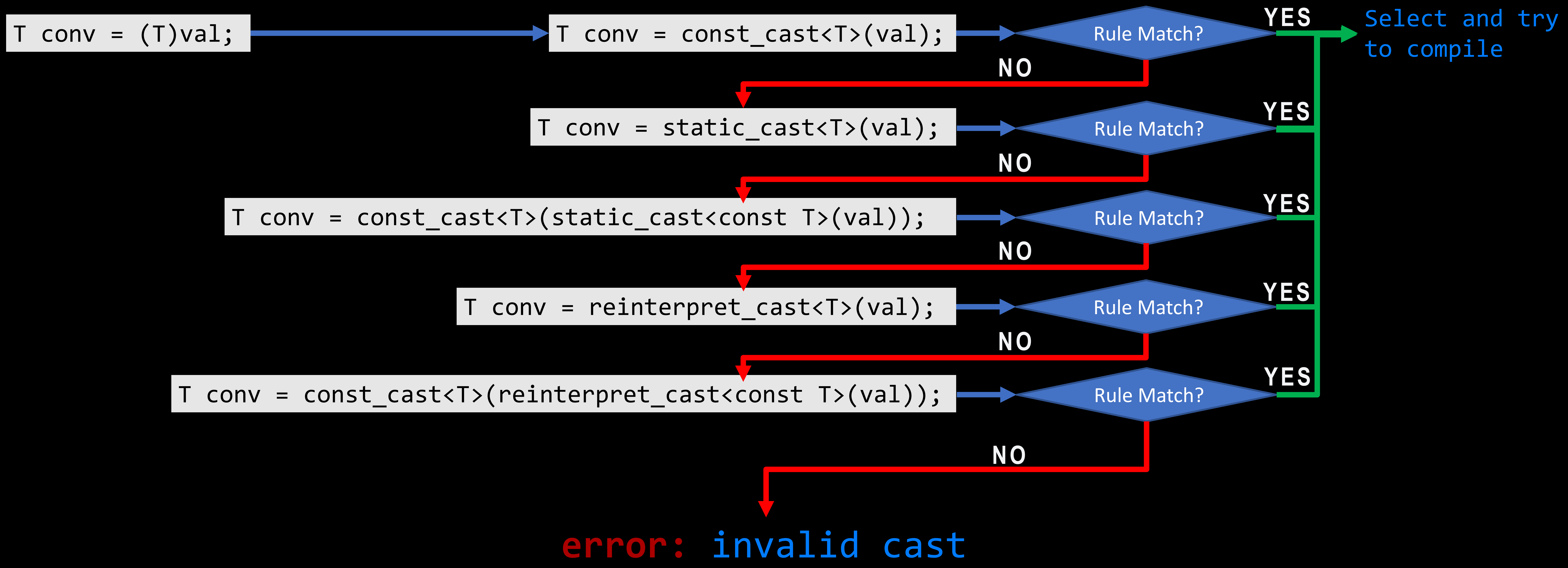
```
Program returned: 0
  px: 1 p3dx: 1
  ap[1]x: 3, ap3d[1]x: 4
```

it depends…

```
Point p{1,2};
auto* p3d =
  reinterpret_cast<Point3D*>(&p);
printf("px: %d p3dx: %d", p.x, p3d->x);

std::array ap = {Point{1,2}, Point{3,4}};
auto* ap3d =
  reinterpret_cast<Point3D*>(ap.data());
printf("ap[1]x: %d, ap3d[1]x: %d", ap[1].x,
  ap3d[1].x);
```

https://tinyurl.com/32b3cdjp

How C-style Casts are *Really* Performed in C++



A selection of additional C++ "casts"

1. `bit_cast`
2. `move/move_if_noexcept`
3. `forward`
4. `as_const`
5. `to_underlying` (C++23)

bit_cast<T>

```
struct A{int x; int y;};
struct B{ double d; };
struct C{ int i; };

int main()
{
 A a;
 B b = std::bit_cast<B>(a);
 //C c = std::bit_cast<C>(a); //different size
 float f = 7.05f;
 uint32_t ui = std::bit_cast<uint32_t>(f);
 return 0;
}
```

1. Located in the <bit> header
2. Converts From into a bit representation in To
3. Requires To and From to be the same size
4. Requires To and From to be trivially copyable
5. Can be used in a constexpr context*
6. Fails to compile if cast is invalid
7. Can introduce UB
8. Requires C++20

*As long as To and From are both not or do not contain: union, pointer, pointer to member, volatile-qualified type or have a non-static data member of reference type

bit_cast<T>

If you need this type of cast in C or pre C++20, use memcpy not a cast:

```
int main()
{
 float f = 7.05f;
 uint32_t ui = 0;
 memcpy(&ui, &f, sizeof(f));
 return 0;
}
```

```
struct A{int x; int y;};
str
struct C{ int i; };

int main()
{
 A a;
 B b = std::bit_cast<B>(a);
 //C c = std::bit_cast<C>(a); //different siz
 float f = 7.05f;
 uint32_t ui = std::bit_cast<uint32_t>(f);
 return 0;
}
```

1. Located in the <bit> header
2. Converts From into a bit representation in To
3. …ame size
4. …ly copyable
5. …ntext*

*As long as To and From are both not or do not contain: union, pointer, pointer to member, volatile-qualified type or have a non-static data member of reference type

std::move & std::move_if_no_except

```
struct Employee {
    Employee(std::string aName)
      : mName(std::move(aName)) {}

    private:
     std::string mName;
};

int main()
{
 std::vector<std::unique_ptr<Employee>> employees;
 auto person = std::make_unique<Employee>("me");
 employees.emplace_back(std::move(person));
 return 0;
}
```

1. Converts named variables (lvalues) to unnamed variables (rvalues)
2. If present, calls the move constructor of To, if not, calls copy constructor
3. Do not return `std::move(var)`
4. Equivalent to `static_cast<T&&>(var)`
5. `move_if_no_except` will trigger the copy constructor if the move constructor of the destination is not marked `noexcept`

std::forward

```
void f(int const &arg) { puts("by lvalue"); }
void f(int && arg) { puts("by rvalue"); }

template< typename T >
void func(T&& arg) {
    printf(" std::forward: ");
    f( std::forward<T>(arg) );
    printf(" normal: ");
    f(arg);
}
```

- Keeps the rvalue or lvalued-ness type passed to a template when passing to another function

std::forward

```
void f(int const &arg) { puts("by lvalue"); }
void f(int && arg) { puts("by rvalue"); }

template< typename T >
void func(T&& arg) {
    printf(" std::forward: ");
    f( std::forward<T>(arg) );
    printf(" normal: ");
    f(arg);
}

int main() {
    puts("call with rvalue:");
    func(5);
```

- Keeps the rvalue or lvalued-ness type passed to a template when passing to another function

```
call with rvalue:
 std::forward: by rvalue
 normal: by lvalue
```

std::forward

```
void f(int const &arg) { puts("by lvalue"); }
void f(int && arg) { puts("by rvalue"); }

template< typename T >
void func(T&& arg) {
    printf(" std::forward: ");
    f( std::forward<T>(arg) );
    printf(" normal: ");
    f(arg);
}

int main() {
    puts("call with rvalue:");
    func(5);
    puts("call with lvalue:");
    int x = 5;
    func(x);
}
```

- Keeps the rvalue or lvalued-ness type passed to a template when passing to another function

```
call with rvalue:
 std::forward: by rvalue
 normal: by lvalue
call with lvalue:
 std::forward: by lvalue
 normal: by lvalue
```

std::as_const

```
struct S {
  void f() const { puts("const"); }
  void f() { puts("non-const"); }
};

int main() {
  S s;
  s.f();
  std::as_const(s).f();
  return 0;
}
```

```
Program returned: 0
     non-const
     const
```

1. Adds `const` to var
2. Less verbose way of doing `static_cast<const T&>(var)`

std::to_underlying

```
enum struct Result : int16_t { Ok = 1 };

enum Unscoped : int { Fail = -1 };

void print(int i) {
 std::cout << "int: " << i << "\n";
}
void print(int16_t i) {
  std::cout << "int16_t: " << i << "\n";
}

int main() {
 auto res = Result::Ok;
 print(std::to_underlying(res));
 auto unscoped = Fail;
 print(std::to_underlying(unscoped));
 return 0;
}
```

1. Proposed for C++23
2. Converts a sized enum to its value as the underlying type
3. Equivalent to
   `static_cast<std::underlying_type_t<T>>(var)`

```
Program returned: 0
     int16_t: 1
     int: -1
```

Which cast to use?

Which cast to use?

Ask yourself this...	... and do this
Can I use the correct type and not cast?	Change types and do not cast

Which cast to use?

Ask yourself this...	... and do this
Can I use the correct type and not cast?	Change types and do not cast
Does it compile with `static_cast` and you know the original variable type?	Use `static_cast`

Which cast to use?

Ask yourself this...	... and do this
Can I use the correct type and not cast?	Change types and do not cast
Does it compile with `static_cast` and you know the original variable type?	Use `static_cast`
Do I not know the original variable type and want to check if a pointer or reference to base class was originally some derived type?	Use `dynamic_cast` and check for `nullptr` or handle `std::bad_cast`

Which cast to use?

Ask yourself this...	... and do this
Can I use the correct type and not cast?	Change types and do not cast
Does it compile with `static_cast` and you know the original variable type?	Use `static_cast`
Do I not know the original variable type and want to check if a pointer or reference to base class was originally some derived type?	Use `dynamic_cast` and check for `nullptr` or handle `std::bad_cast`
Is my original variable modifiable or is it `const` and won't be modified?	Use `const_cast` or `as_const`

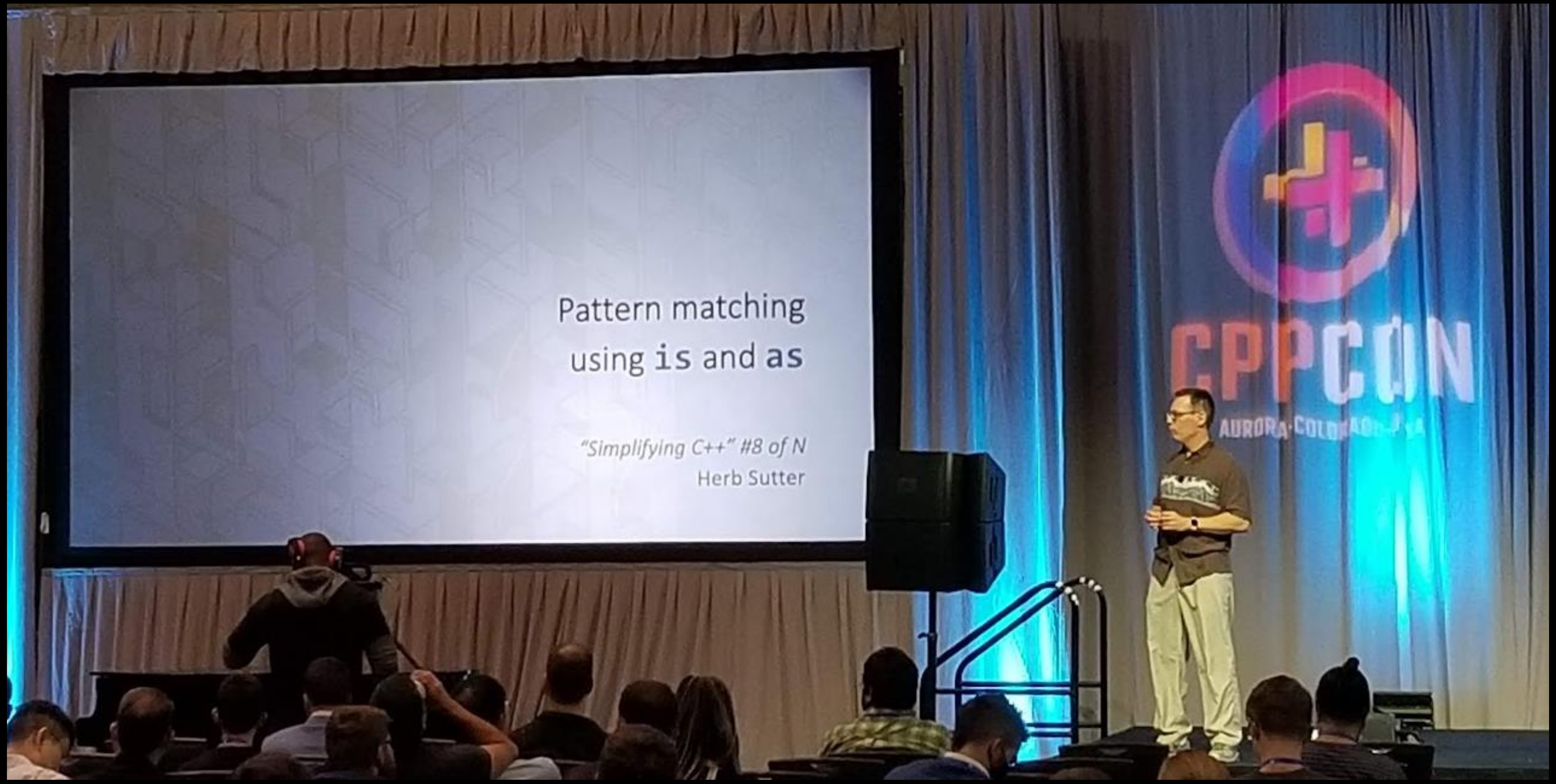
Which cast to use?

Ask yourself this…	… and do this
Can I use the correct type and not cast?	Change types and do not cast
Does it compile with `static_cast` and you know the original variable type?	Use `static_cast`
Do I not know the original variable type and want to check if a pointer or reference to base class was originally some derived type?	Use `dynamic_cast` and check for `nullptr` or handle `std::bad_cast`
Is my original variable modifiable or is it `const` and won't be modified?	Use `const_cast` or `as_const`
Do I want to examine the bits of a type using a different type of the same size?	Use `bit_cast (C++20 only)`

Which cast to use?

Ask yourself this...	... and do this
Can I use the correct type and not cast?	Change types and do not cast
Does it compile with `static_cast` and you know the original variable type?	Use `static_cast`
Do I not know the original variable type and want to check if a pointer or reference to base class was originally some derived type?	Use `dynamic_cast` and check for `nullptr` or handle `std::bad_cast`
Is my original variable modifiable or is it `const` and won't be modified?	Use `const_cast or as_const`
Do I want to examine the bits of a type using a different type of the same size?	Use `bit_cast (C++20 only)`
Do I know exactly what is in memory and need to treat it differently?	Use `reinterpret_cast`

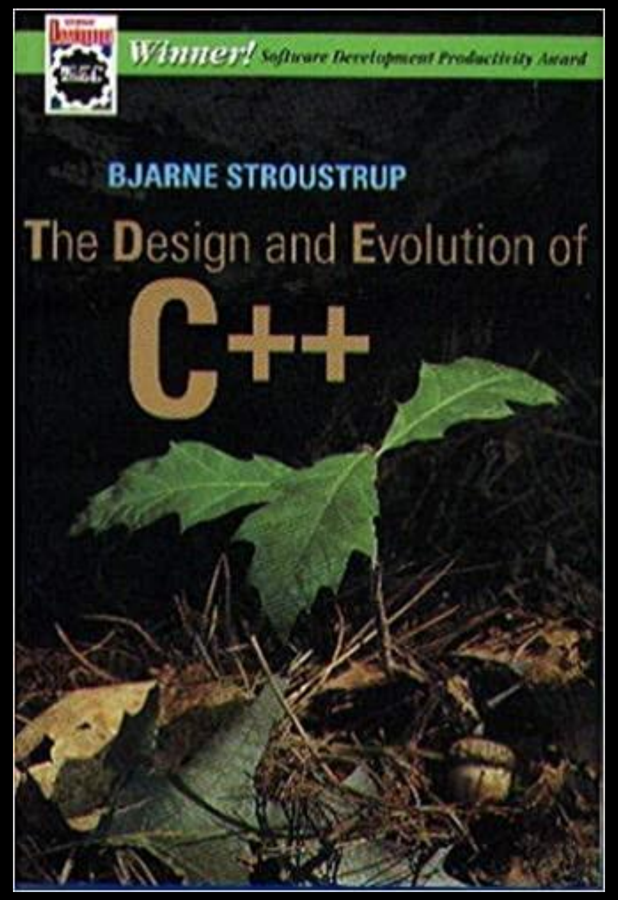
A possible new casting syntax



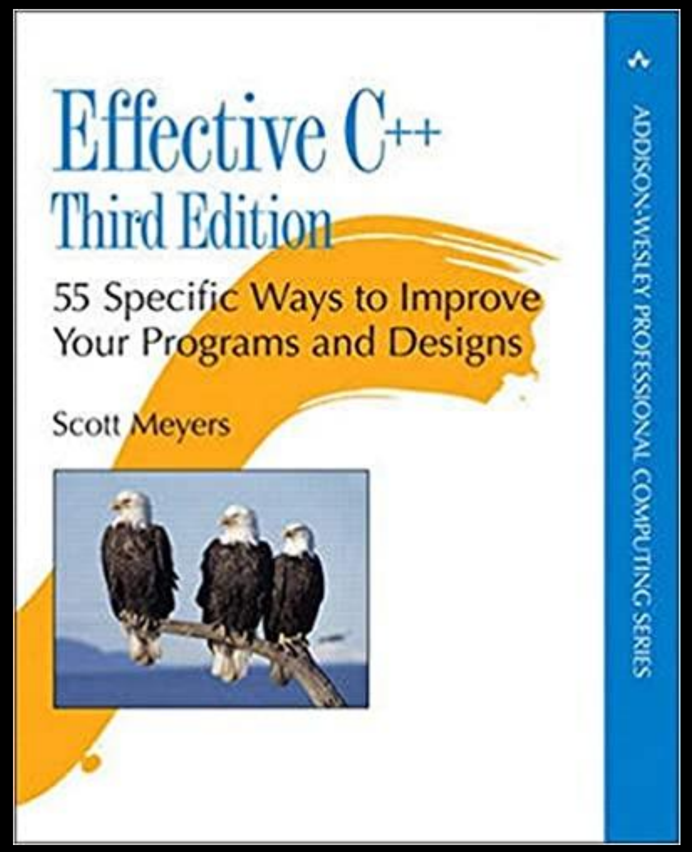
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2392r0.pdf

Resources:

- Bjarne Stroustrup - The Design and Evolution of C++



- Scott Meyers – Effective C++ (3rd Edition)



Back To Basics }THANKS

Casting

GARMIN.