TMP: template meta-programming (I)

Deep dive into C++

scc@teamt5.org

TEAMT5
杜浦數位安全
Persistent Cyber Threat Hunters

Outline

TEAMT5 杜浦數位安全

Template 101 and type deduction

Perfect forwarding and reference collapsing

Variadic template and parameter pack

SFINAE

CTAD

Expression template

Concepts and constraints

CRTP and compile time polymorphism

Template 101 & type deduction

Generic type?

```
int square(int num) {
    return num * num;
}
```

```
template<typename T>
T square(T num) {
    return num * num;
}
```

_Generic (C11)

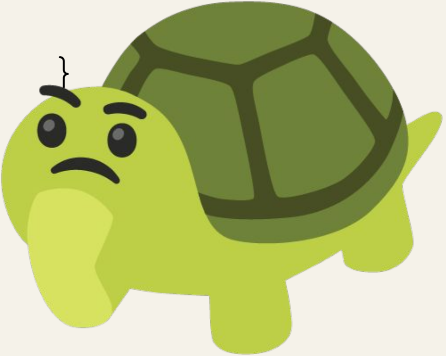
```c
// Specific square functions for different types
int square_int(int num) {
    return num * num;
}
double square_double(double num) {
    return num * num;
}
```

```c
#define square(X) _Generic((X), \
    int: square_int,           \
    double: square_double,     \
    default: square_int        \
)(X)
```

_Generic (C11)

```
// Spe                                    ifferent types
int s
    re
}
double sq      _double(double num) {
    return num * num;
}
```

```
#define square(X) _Generic((X), \
    int: square_int,            \
    double: square_double,      \
    default: square_int         \
)(X)
```



1. Hand-write name mangling
2. Lots of types
3. C89 you should write the specific version

Template 101

```
template<typename T>
T max(T a, T b) {
    return a > b ? a : b;
}
```

Template 101

```
template<parameter-list >
```

```
template<typename T>
T max(T a, T b) {
    return a > b ? a : b;
}
```

Template 101

```
template<parameter-list >
```

```
template<typename T>
T max(T a, T b) {
    return a > b ? a : b;
}
```

Type

Non-type

Template

Template 101

```
template<parameter-list >
```

Type

Non-type

Template

```
T max(T a, T b) {
    return a > b ? a : b;
}
```

Template 101

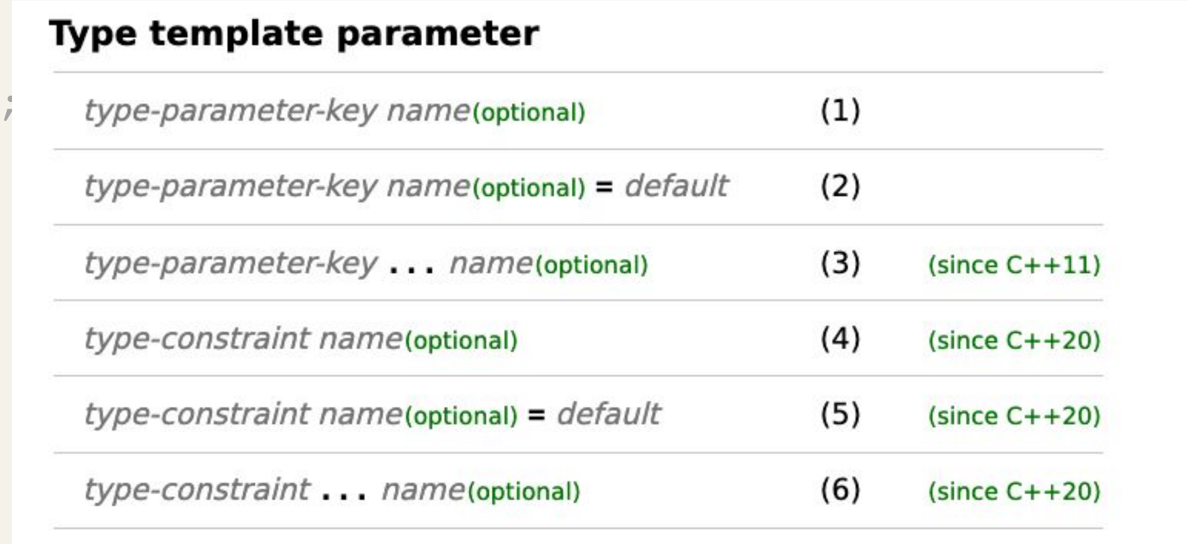
```
template<parameter-list >
```

Type

Non-type

Template

```
y
T max(T a, T b) {
    return a > b ? a : b;
}
```



Type template parameter

type-parameter-key name(optional)	(1)	
type-parameter-key name(optional) = *default*	(2)	
type-parameter-key ... *name*(optional)	(3)	(since C++11)
type-constraint name(optional)	(4)	(since C++20)
type-constraint name(optional) = *default*	(5)	(since C++20)
type-constraint ... *name*(optional)	(6)	(since C++20)

Template 101

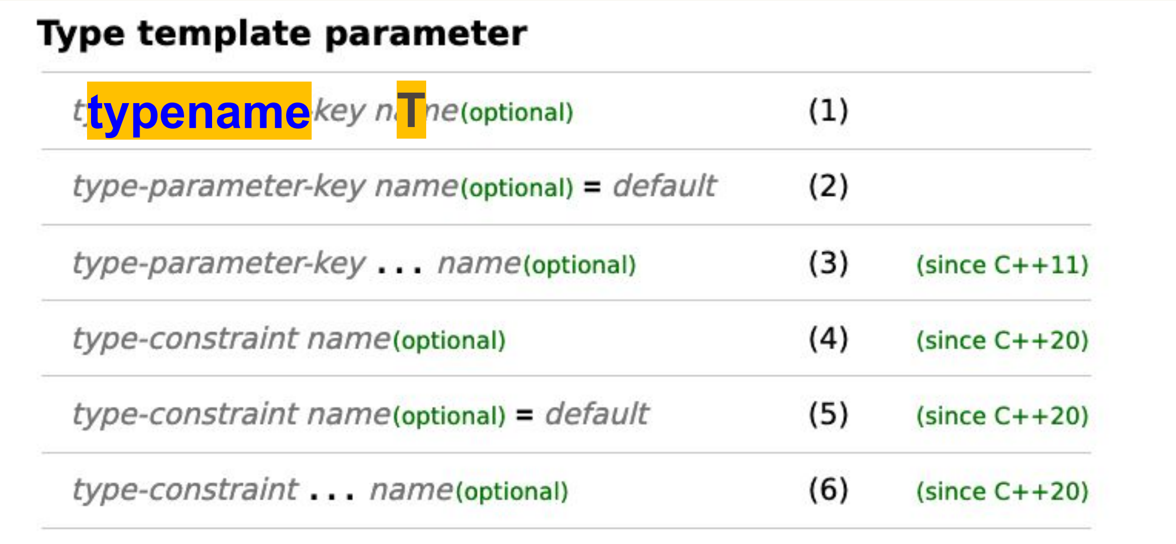
```
template<parameter-list >
```

Type

Non-type

Template

```
T max(T a, T b) {
    return a > b ? a : b;
}
```



Type template parameter

t**typename**key n**T**ne(optional)	(1)	
type-parameter-key name(optional) = *default*	(2)	
type-parameter-key ... *name*(optional)	(3)	(since C++11)
type-constraint name(optional)	(4)	(since C++20)
type-constraint name(optional) = *default*	(5)	(since C++20)
type-constraint ... *name*(optional)	(6)	(since C++20)

Template 101

```
template<parameter-list >
```

Type

Non-type

Template

```
T max(T a, T b) {
    return a > b ? a : b;
}
```

Template 101

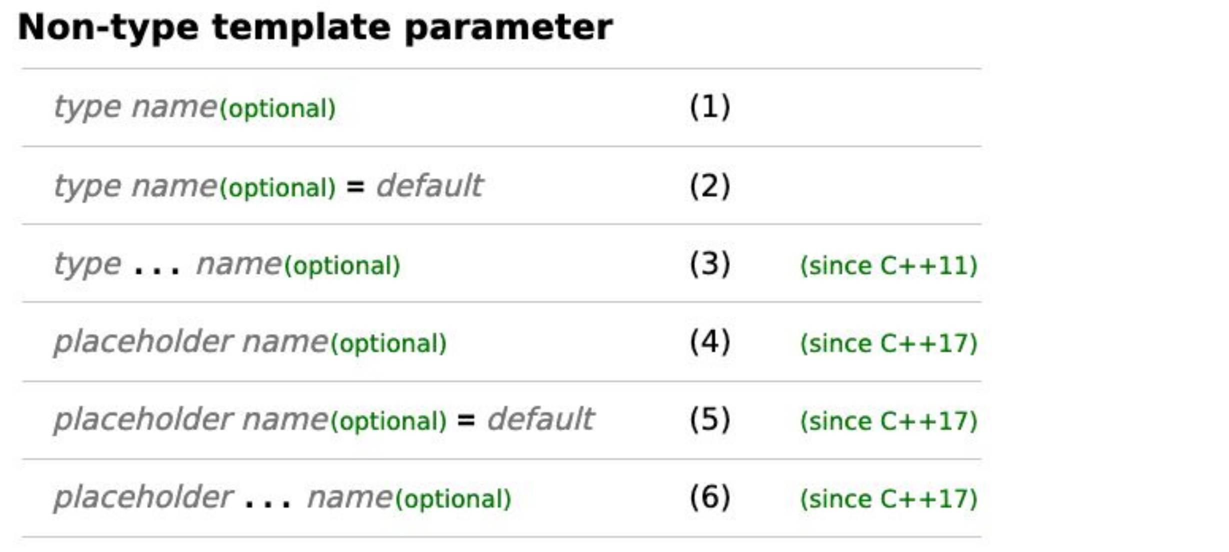
```
template<parameter-list >
```

Type

Non-type

Template

```
T max(T a, T b) {
    return a > b ? a :
}
```



Non-type template parameter

type name(optional)	(1)	
type name(optional) = *default*	(2)	
type ... *name*(optional)	(3)	(since C++11)
placeholder name(optional)	(4)	(since C++17)
placeholder name(optional) = *default*	(5)	(since C++17)
placeholder ... *name*(optional)	(6)	(since C++17)

Template 101

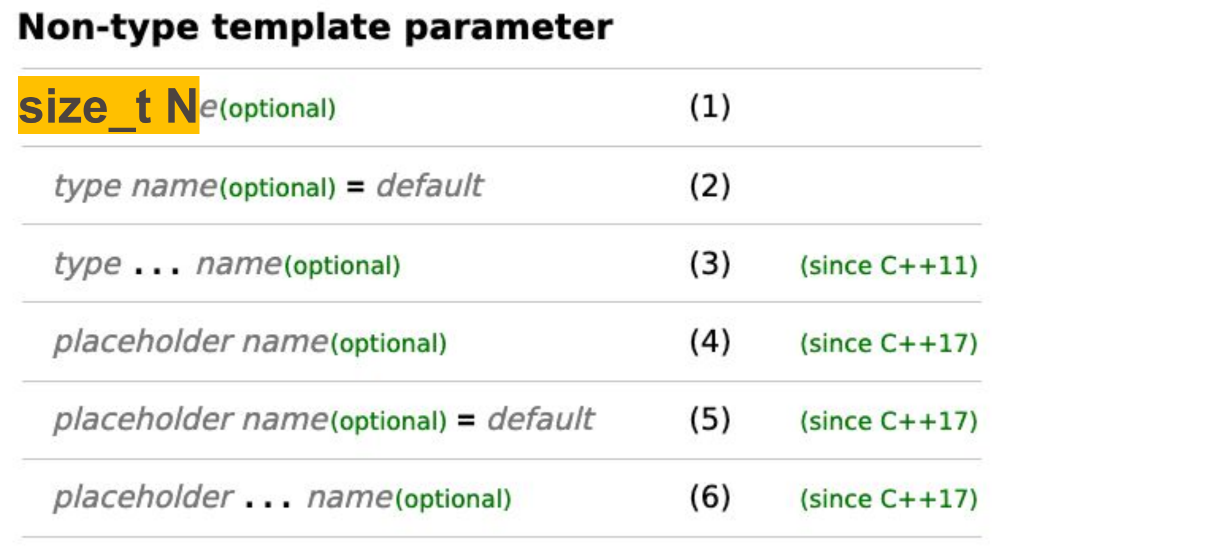
```
template<parameter-list >
```

Type

Non-type

Template

```
T max(T a, T b) {
    return a > b ? a :
}
```



Non-type template parameter

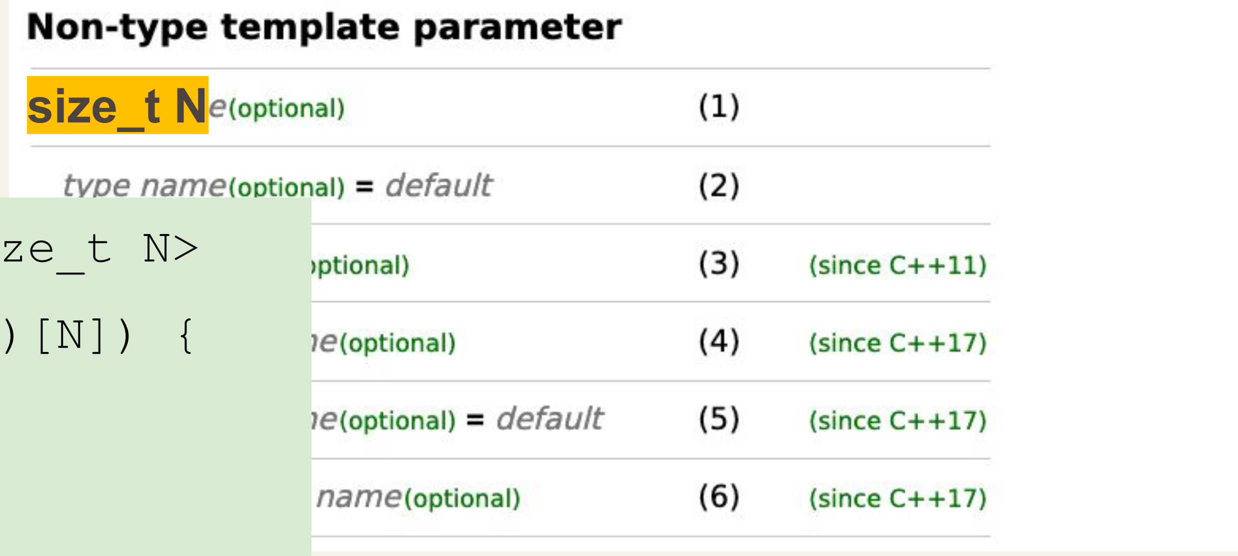
size_t Ne(optional)	(1)	
type name(optional) = *default*	(2)	
type ... *name*(optional)	(3)	(since C++11)
placeholder name(optional)	(4)	(since C++17)
placeholder name(optional) = *default*	(5)	(since C++17)
placeholder ... *name*(optional)	(6)	(since C++17)

Template 101

```
template<parameter-list >
```

Type | Non-type | Template

```
T max(T a, T b) {
    return a > b ? a :
```



Non-type template parameter

size_t Ne(optional)	(1)	
type name(optional) = default	(2)	
ptional)	(3)	(since C++11)
ne(optional)	(4)	(since C++17)
ne(optional) = default	(5)	(since C++17)
name(optional)	(6)	(since C++17)

```
template<typename T, size_t N>
size_t strlen(const T(&)[N]) {
    return N - 1;
}
```

Template 101

```
template<parameter-list >
```

Type

Non-type

Template

```
T max(T a, T b) {
    return a > b ? a : b;
}
```

Template 101

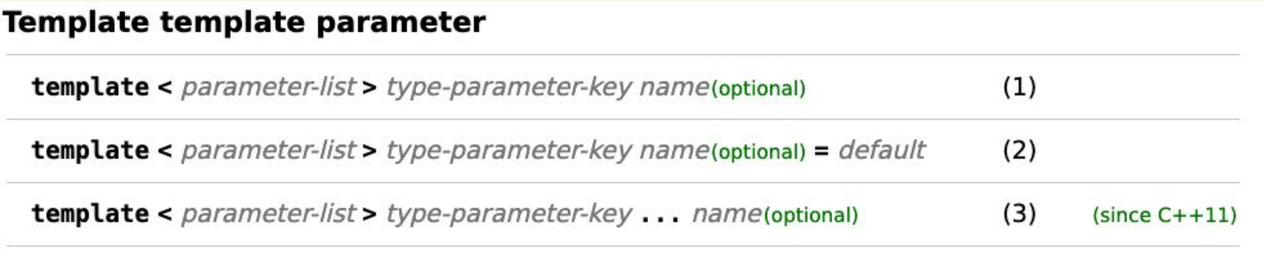
template<parameter-list >

Type

Non-type

Template

```
T max(T a, T b) {
    return a > b ? a :
}
```



Template template parameter

template < *parameter-list* > *type-parameter-key name*(optional)	(1)	
template < *parameter-list* > *type-parameter-key name*(optional) = *default*	(2)	
template < *parameter-list* > *type-parameter-key* **...** *name*(optional)	(3)	(since C++11)

Template 101

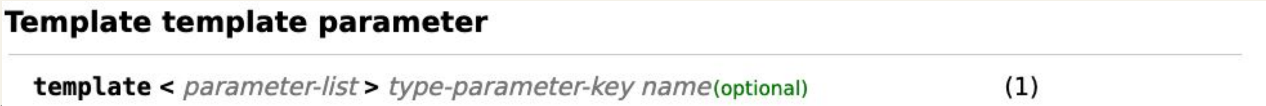
template<parameter-list >

Type

Non-type

Template

```
T max(T a, T b) {
    return a > b ? a
```



Template template parameter

template < *parameter-list* **>** *type-parameter-key name*(optional) (1)

```
template<typename T, template<typename> typename Container>
void print(Container<T> v) {
    for (const auto& item : v)
        std::cout << item << " ";
}
```

Template specialization

```
std::vector<bool>
```

```
template<typename T>
class Vector {
    std::vector<T> data;

public:
    void push_back(const T& v) { data.push_back(v); }
    void push_back(T&& v) { data.push_back(std::forward<T>(v)); }
    const T& at(size_t pos) const { return data[pos]; }
    T& at(size_t pos) { return data[pos]; }
    size_t size() const { return data.size(); }
};
```

https://godbolt.org/z/649aYfTvz

```
template<>
class Vector<bool> {
    using T = bool;
    std::vector<char> data;

public:
    void push_back(const T& v) { data.push_back(v); }
    void push_back(T&& v) { data.push_back(std::forward<T>(v)); }
    bool at(size_t pos) const { return data[pos]; }
    size_t size() const { return data.size(); }
};
```

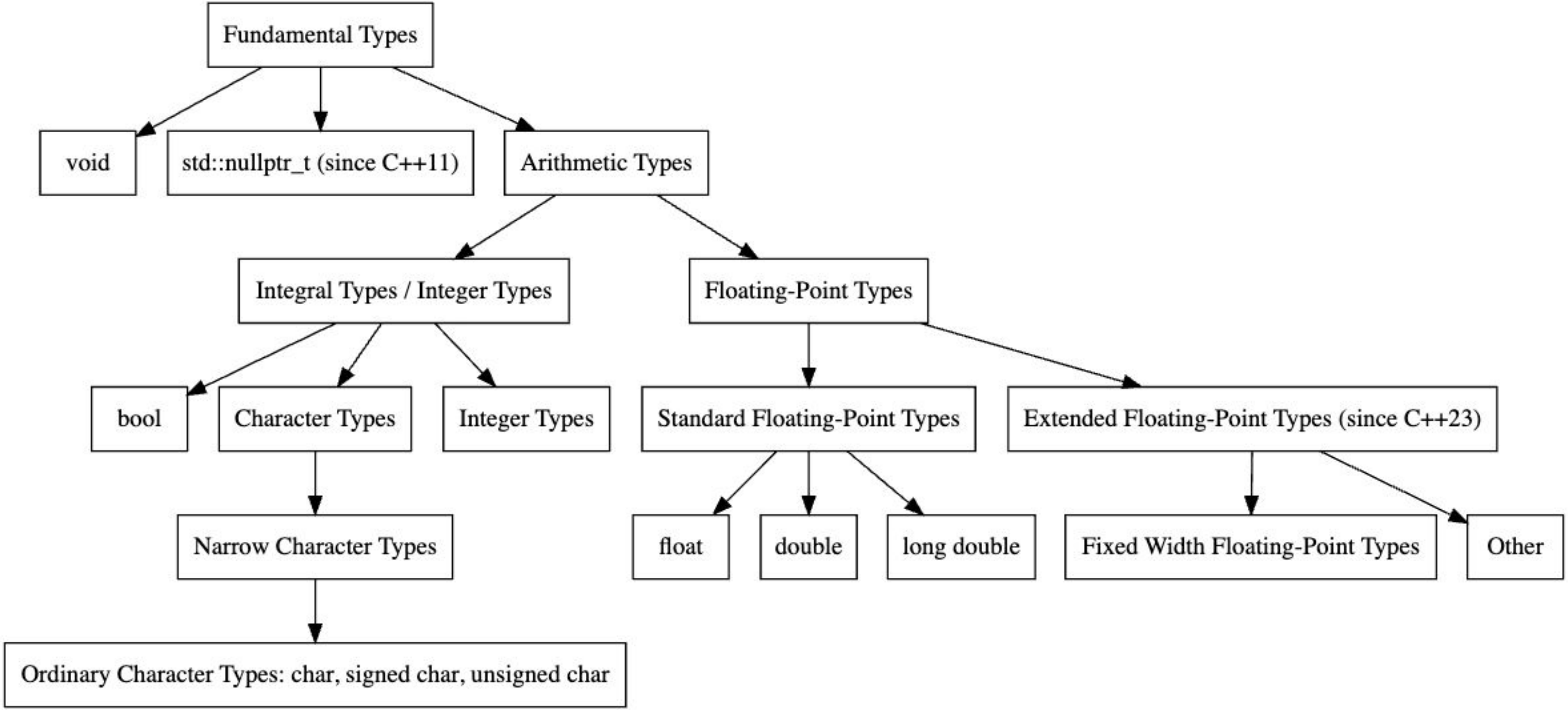
https://godbolt.org/z/649aYfTvz

Type deductions

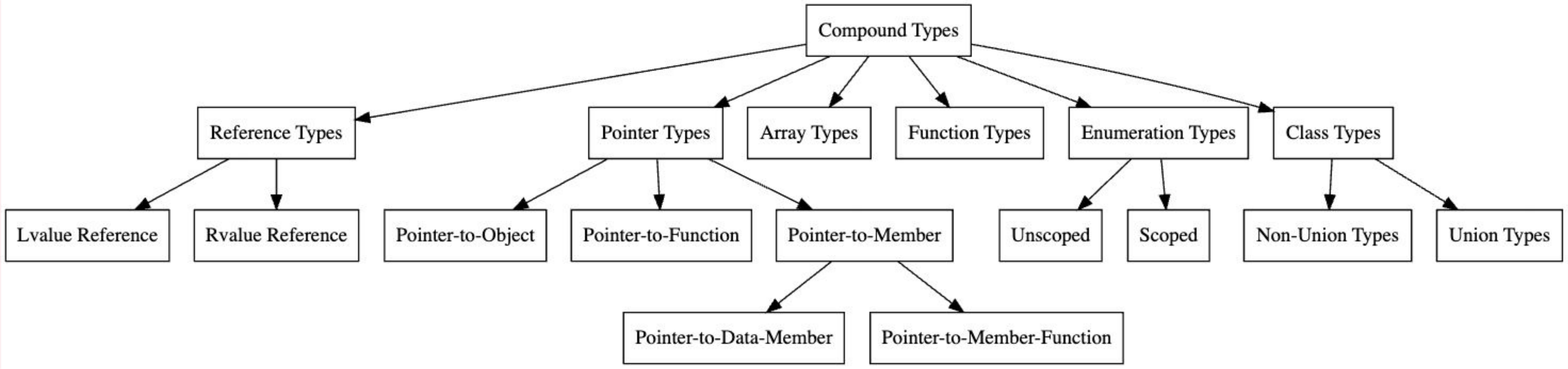
Fundamental types

graphiz

TEAMT5 杜浦數位安全


Fundamental Types
void
std::nullptr_t (since C++11)
Arithmetic Types
Integral Types / Integer Types
Floating-Point Types
bool
Character Types
Integer Types
Standard Floating-Point Types
Extended Floating-Point Types (since C++23)
Narrow Character Types
float
double
long double
Fixed Width Floating-Point Types
Other
Ordinary Character Types: char, signed char, unsigned char

Compound types



graphiz

頭腦體操

```
print_type<decltype(1 + 1ull)>();       // unsigned long long
print_type<decltype(1. + 1)>();         // double
print_type<decltype(+ -+-+-+ +true)>(); // int
print_type<decltype([]{})>();           // lambda type
print_type<decltype("dummy")>();        // const char (&) [6]
print_type<decltype(struct {})>();      // (anonymous struct)
print_type<void (*)()>();               // void (*)()
print_type<decltype(nullptr)>();        // std::nullptr_t
print_type<decltype(3.14f * 'a')>();    // float
print_type<decltype(0b1010 + 0xFF)>();  // int
print_type<decltype(~~~0u)>();          // unsigned int
print_type<decltype("C++Expert"[3])>(); // const char &
```

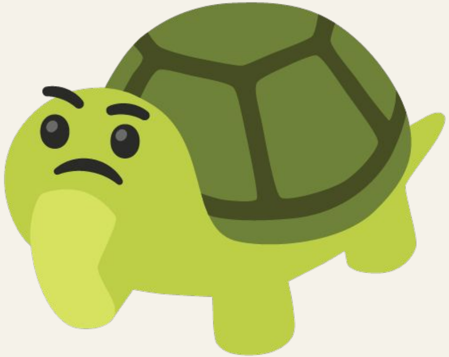
https://godbolt.org/z/Tazrnfzd7

So, it’s easy

```
template<typename T>
void print_type(){
    auto me = std::string(__PRETTY_FUNCTION__);
    // ...
}
```

So, it’s easy

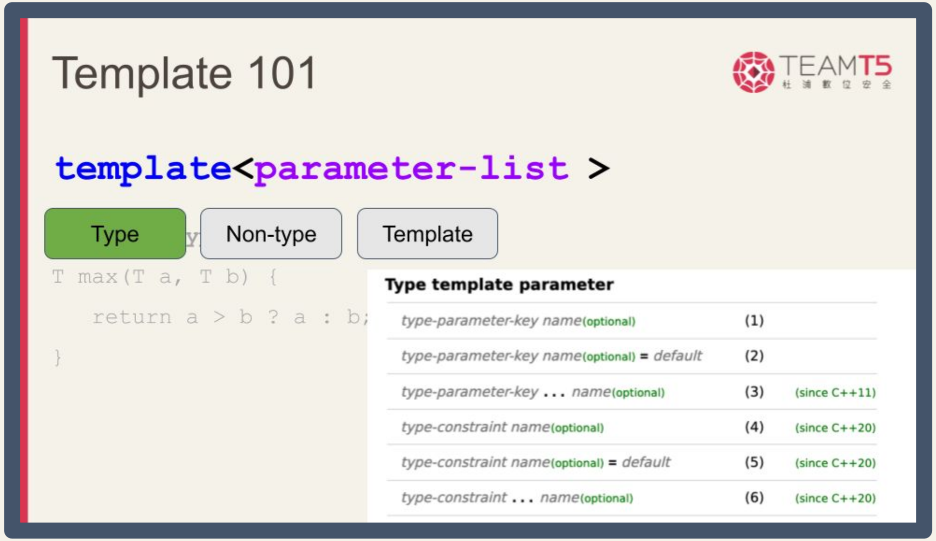
```
void print_type(auto v){
    auto me = std::string(__PRETTY_FUNCTION__);
    // ...
}
```



C++20
syntactic sugar

Deducting type from Args

```
template<typename Para>
void f(Para param);


f(expr);
```



Template 101

```
template<parameter-list >
```

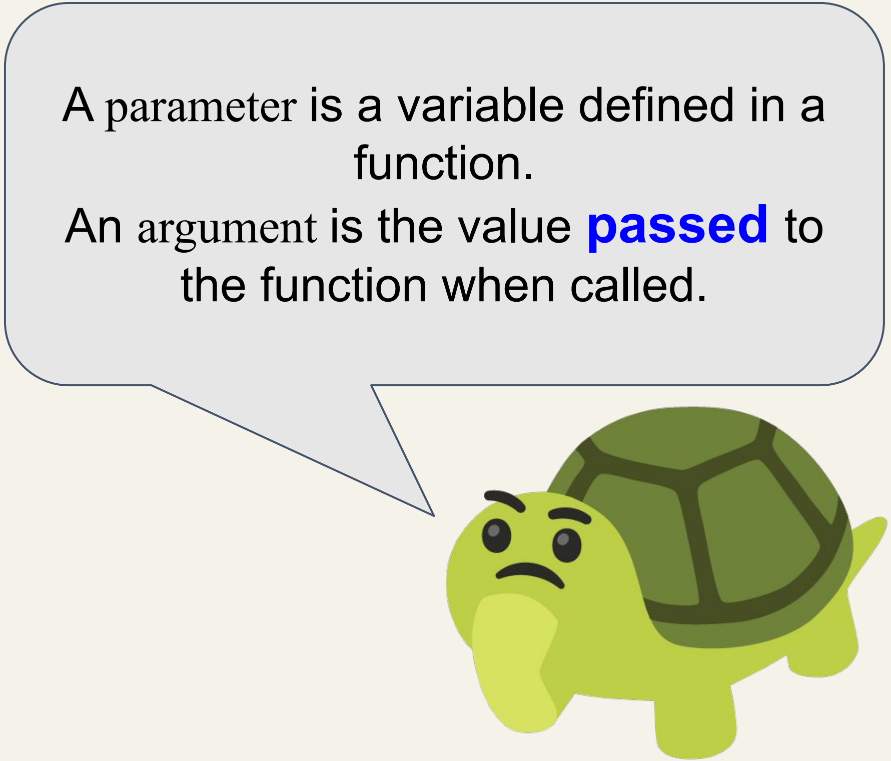
```
T max(T a, T b) {
   return a > b ? a : b;
}
```

Type template parameter

type-parameter-key name(optional)	(1)	
type-parameter-key name(optional) = *default*	(2)	
type-parameter-key ... *name*(optional)	(3)	(since C++11)
type-constraint name(optional)	(4)	(since C++20)
type-constraint name(optional) = *default*	(5)	(since C++20)
type-constraint ... *name*(optional)	(6)	(since C++20)

Deducting type from Args

```
template<typename Para>
void f(Para param);


f(expr);
```



A parameter is a variable defined in a function.

An argument is the value **passed** to the function when called.

Deducting type from Args

```
template<typename Para>
void f(Para param);


f(expr);
```

A parameter is a variable defined in a function.

An argument is the value **passed** to the function when called.

Deducing Args type and making Para being defined.

Deducting type from Args

```
template<typename Para>
void f(Para param);


f(expr);
```

A parameter is a variable defined in a function.

An argument is the value **passed** to the function when called.

Deducing Args type and making Para being defined.

Template
Argument
Deduction

template type

```
template<typename Para>
void f(Para param);


f(expr);
```

Regular

Reference

Universal/Forwarding reference

template type

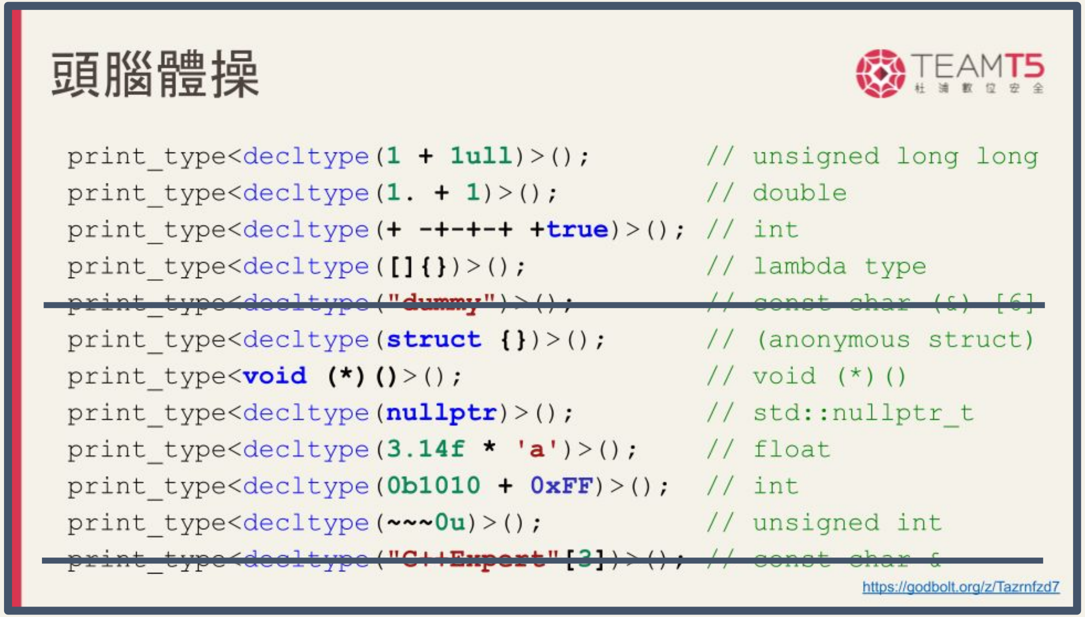
- Regular
- Reference
- Universal/Forwarding reference

template type

Regular

Reference

Universal/Forwarding reference



頭腦體操

```
print_type<decltype(1 + 1ull)>();         // unsigned long long
print_type<decltype(1. + 1)>();           // double
print_type<decltype(+ -+-+-+ +true)>();   // int
print_type<decltype([]{})>();             // lambda type
~~print_type<decltype("dummy")>();         // const char (&) [6]~~
print_type<decltype(struct {})>();        // (anonymous struct)
print_type<void (*)()>();                 // void (*)()
print_type<decltype(nullptr)>();          // std::nullptr_t
print_type<decltype(3.14f * 'a')>();      // float
print_type<decltype(0b1010 + 0xFF)>();    // int
print_type<decltype(~~~0u)>();            // unsigned int
~~print_type<decltype("C++Expert"[3])>(); // const char &~~
```

https://godbolt.org/z/Tazrnfzd7

Regular

Ignore the cv-qualifier and any references. Copy it anyway.

```
template<typename T>

void f(T param);



int x = 27;
const int cx = x;
const int& rx = x;

f(x);  // T's and param's types are both int
f(cx); // T's and param's types are again both int
f(rx); // T's and param's types are still both int
```

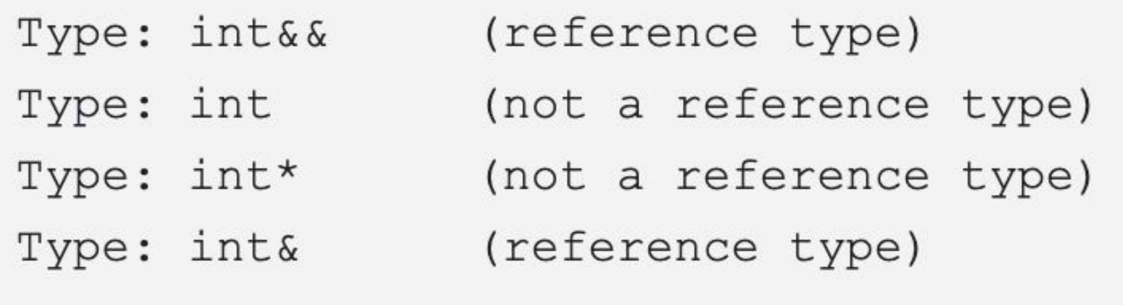
template type

Regular

Reference

Universal/Forwarding reference

```
int main() {
  print_type<decltype(std::move(1))>();
  print_type<decltype(1)>();

  int a = 42;
  print_type<decltype(&a)>();
  print_type<int&>();
//print_type<std::unwrap_reference_t<
              decltype(std::ref(a))>>();
}
```



```
Type: int&&        (reference type)
Type: int          (not a reference type)
Type: int*         (not a reference type)
Type: int&         (reference type)
```

https://godbolt.org/z/vYE7YKWxK

Reference

ParamType is a Reference or Pointer, but not a Universal Reference

1. If ***expr***'s type is a reference, **ignore** the reference part.
2. Then pattern-match ***expr***'s type against ***ParamType*** to determine T.

```
template<typename T>
void f(T& param);


int x = 27;
const int cx = x;
const int& rx = x;
```

Reference

ParamType is a Reference or Pointer, but not a Universal Reference

1. If ***expr***'s type is a reference, **ignore** the reference part.
2. Then pattern-match ***expr***'s type against ***ParamType*** to determine T.

```
template<typename T>

void f(T& param);

int x = 27;

const int cx = x;

const int& rx = x;
```

```
f(x);   // T is int, param's type is int&
f(cx); // T is const int, param's type is const int&
f(rx); // T is const int, param's type is const int&
```

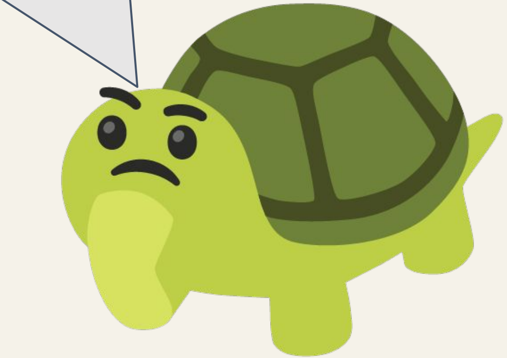
template type

Regular

Reference

Universal/Forwarding reference

We discuss it later. It’s a key point that C++ can be faster than C.



頭腦體操

```
template<typename T>
void F(T param) {}
```

```
int x = 27;
const int cx = x;
const int& rx = x;

F(x);  // T and param's type is int
F(cx); // T and param's type is int (const is ignored)
F(rx); // T and param's type is int (const and ref are ignored)
F(27); // T and param's type is int (rvalue)
```

https://godbolt.org/z/xevoc4ch4

頭腦體操

```
template<typename T>
void F(T param) {}
```

```
const char* const ptr = "Fun with templates";
// T and param's type are const char* (const from pointer itself is ignored)
F(ptr);
// Both T and param's type are const char* (array decays to pointer)
F("temporary string");
```

https://godbolt.org/z/xevoc4ch4

auto type deduction

- Almost the same as template-type deduction
- auto keyword, no longer be a storage specifier, type specifier instead
- auto&& is a universal reference (forwarding reference)

https://www.stroustrup.com/C++11FAQ.html#auto

auto type deduction

```
template<class T>
void printall(const vector<T>& v) {
    for (typename vector<T>::const_iterator p = v.begin(); p!=v.end(); ++p )
        cout << *p << "\n";
}


template<class T>
void printall(const vector<T>& v) {
    for (auto p = v.begin(); p!=v.end(); ++p)
        cout << *p << "\n";
}
```

auto type deduction

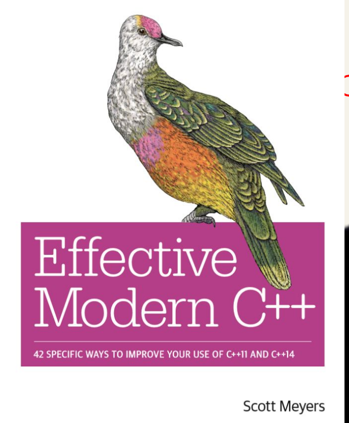
```
#include <initializer_list>
template<typename T>
void foo(T v) {}

int main() {
    auto v = {1, 2, 4}; // OK.
    foo({1, 2, 4});      // template could not deduce the T
}
```

https://godbolt.org/z/h8W5Weha7

auto type deduction

I wonder this myself. Alas, I have not been able to find a convincing explanation. But the rule is the rule…

```
#include <initializer_list>
template<typename T>
void foo(T v) {}

int main() {
    auto v = {1, 2, 4}; // OK.
    foo({1, 2, 4});     // template cou            ce
}
```



https://godbolt.org/z/h8W5Weha7

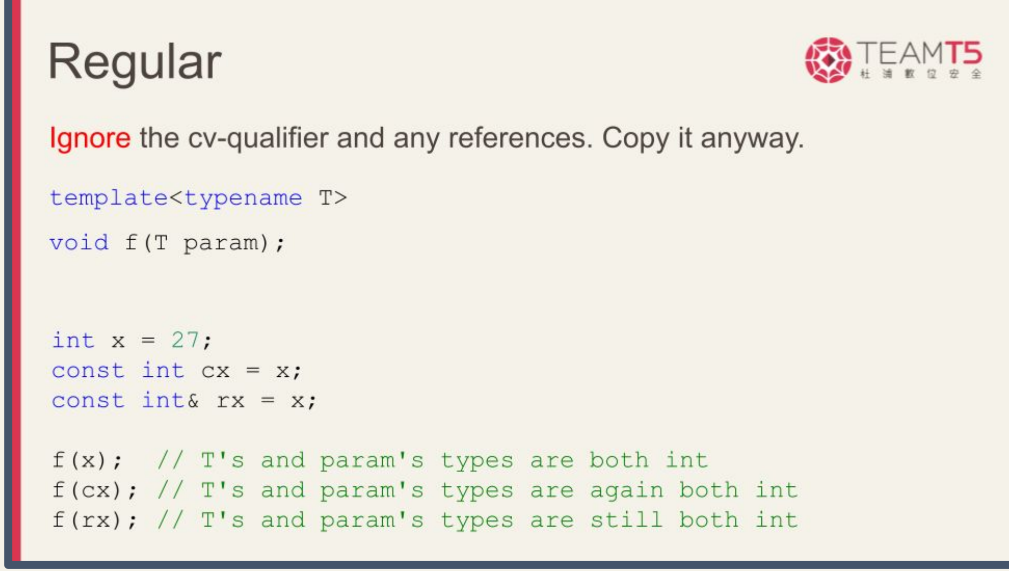
auto type deduction C++14

```
auto foo() { // 14
   return 42;
}

auto foo() -> int { // 11
   return 42;
}
```

decltype(auto)

```
int arr[10];

auto foo() {
    return arr[3]; // return int type
}
```

decltype(auto)



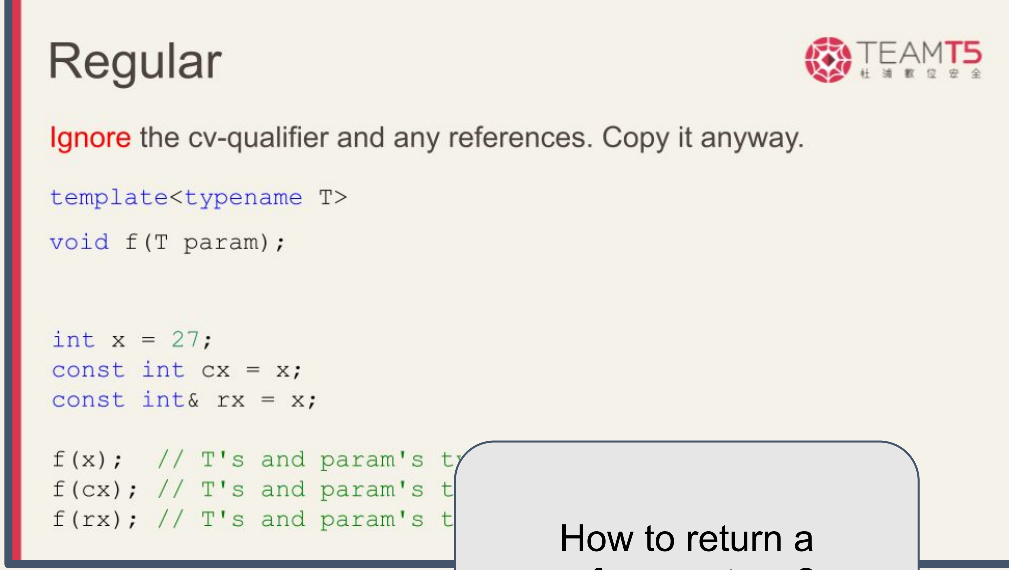
Regular

Ignore the cv-qualifier and any references. Copy it anyway.

```
template<typename T>
void f(T param);


int x = 27;
const int cx = x;
const int& rx = x;

f(x);  // T's and param's types are both int
f(cx); // T's and param's types are again both int
f(rx); // T's and param's types are still both int
```

```
int arr[10];

auto foo() {
   return arr[3]; // return int type
}
```

decltype(auto)



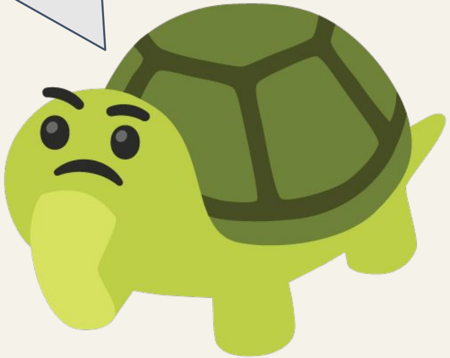
Regular

Ignore the cv-qualifier and any references. Copy it anyway.

```
template<typename T>
void f(T param);


int x = 27;
const int cx = x;
const int& rx = x;

f(x);  // T's and param's ty
f(cx); // T's and param's t
f(rx); // T's and param's t
```

```
int arr[10];

auto foo() {
    return arr[3]; // return int type
}
```

How to return a reference type?



decltype(auto)

```
int arr[10];

auto foo() -> decltype(auto) {
    return arr[3]; // return int type
}
```

decltype(auto)

```
int arr[10];

auto foo() -> decltype(arr[3]) {
    return arr[3]; // return int type
}
```

decltype(auto)

int &

```
int arr[10];

auto foo() -> decltype(arr[3]) {
    return arr[3]; // return int type
}
```

Take away

Template Argument Deduction

- Deducing parameter type from argument
- CV-qualifiers are in the type system

Auto type deduction is similar to TAD, but not initialized_list

- Be careful on curly braces {} initialization

decltype(auto)

- Deduce type from the return expression

Thank you!

scc@teamt5.org

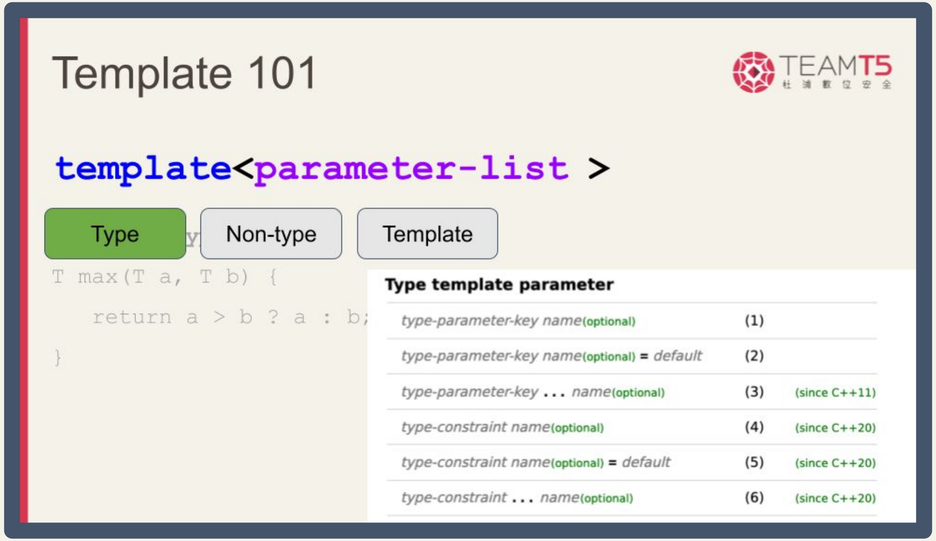
TEAMT5

杜浦數位安全

Persistent Cyber Threat Hunters

Deducting type from Params

```
template<typename ParamType>

void f(ParamType param);


f(expr);
```



Template 101

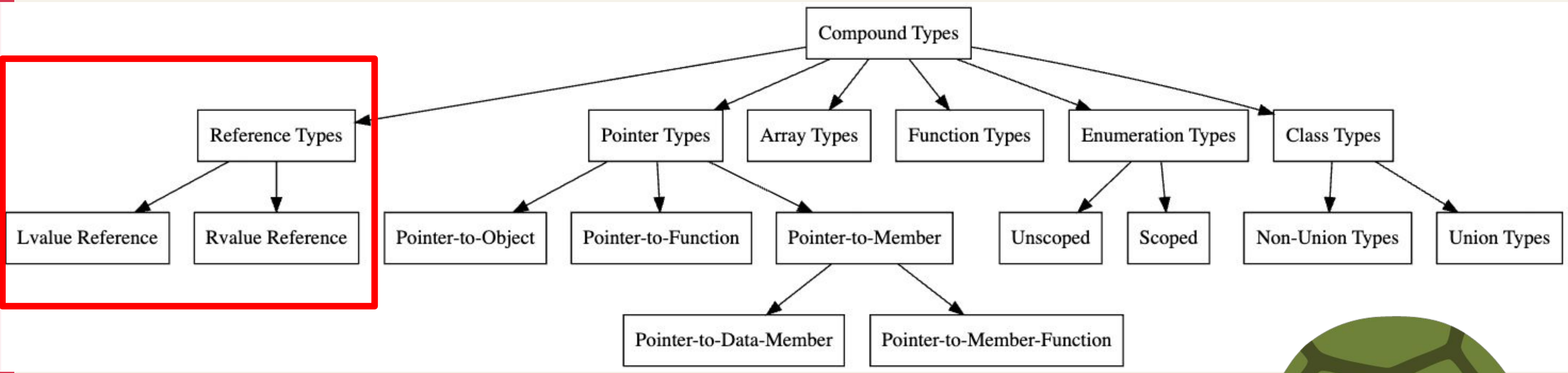
```
template<parameter-list >
```

```
T max(T a, T b) {
   return a > b ? a : b;
}
```

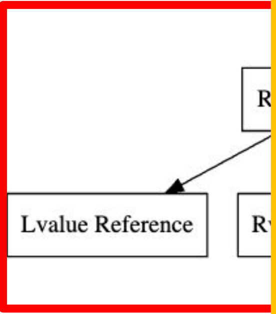
Type template parameter

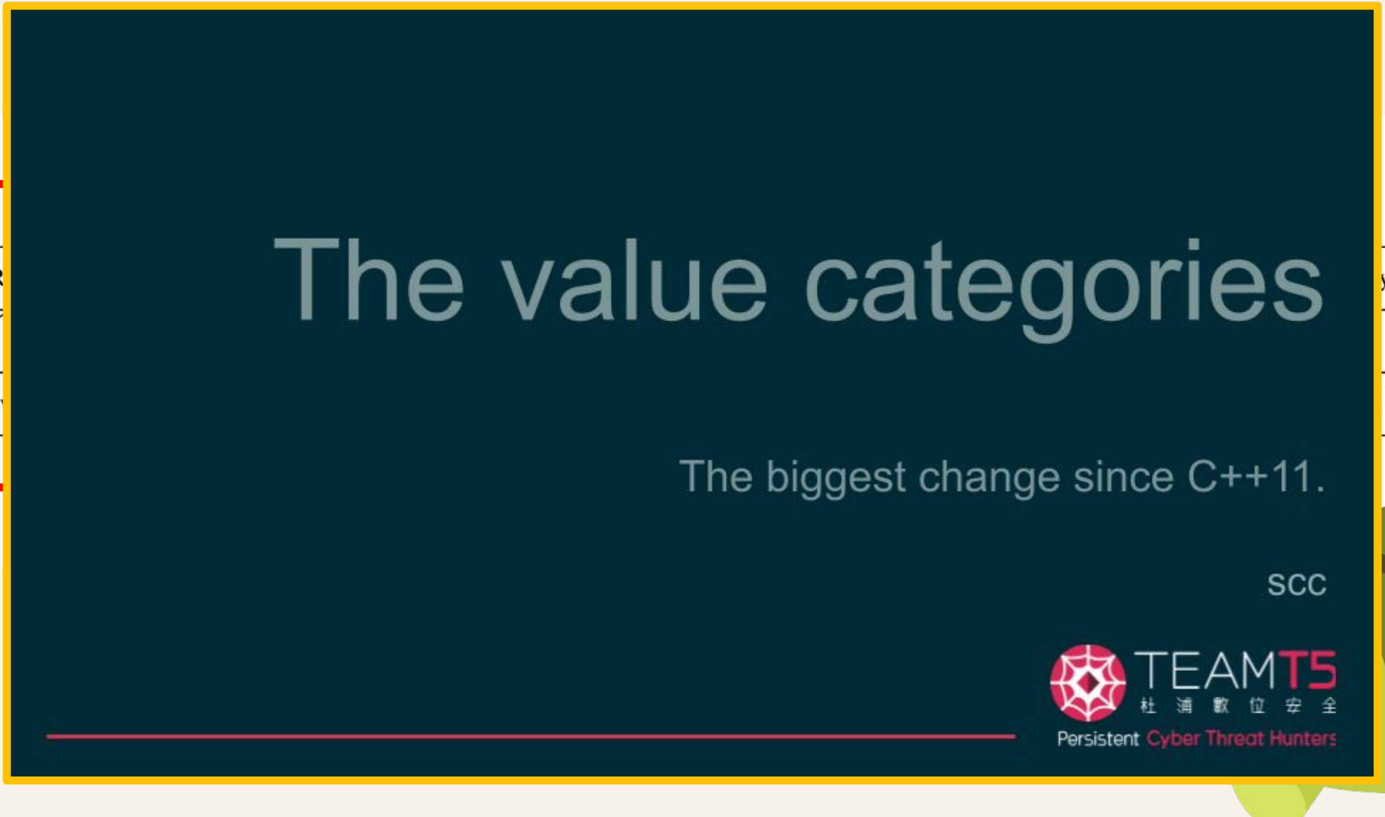
type-parameter-key name(optional)	(1)	
type-parameter-key name(optional) = *default*	(2)	
type-parameter-key ... *name*(optional)	(3)	(since C++11)
type-constraint name(optional)	(4)	(since C++20)
type-constraint name(optional) = *default*	(5)	(since C++20)
type-constraint ... *name*(optional)	(6)	(since C++20)

Reference type



Reference type



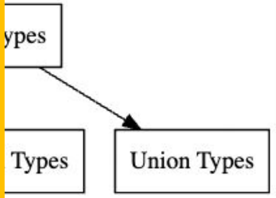


The value categories

The biggest change since C++11.

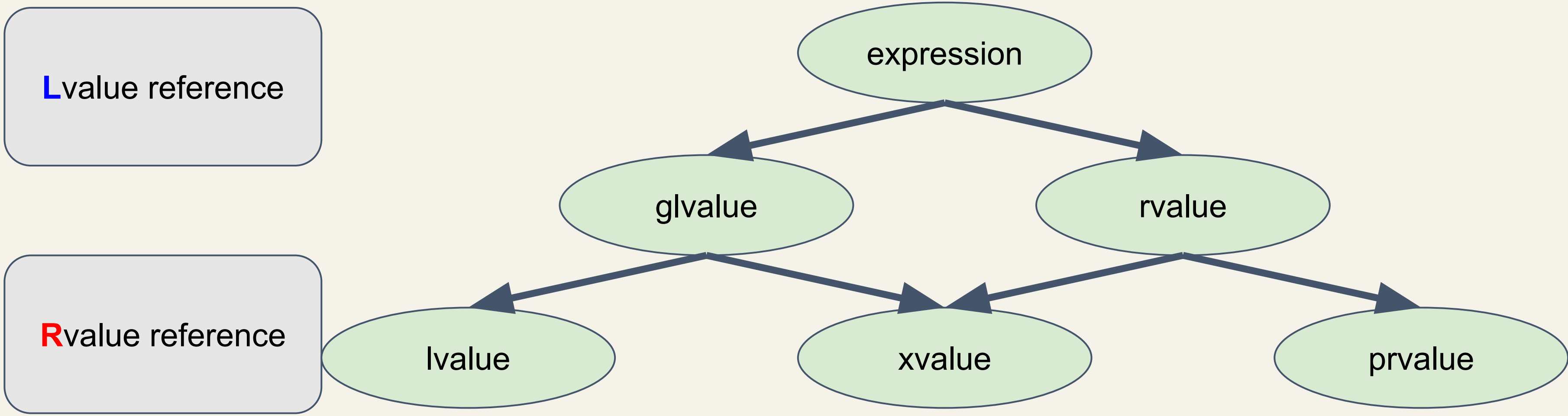
scc

TEAMT5
杜 浦 數 位 安 全
Persistent Cyber Threat Hunters



Reference

Declares a named variable as a reference, that is, an alias to an already-existing object or function.

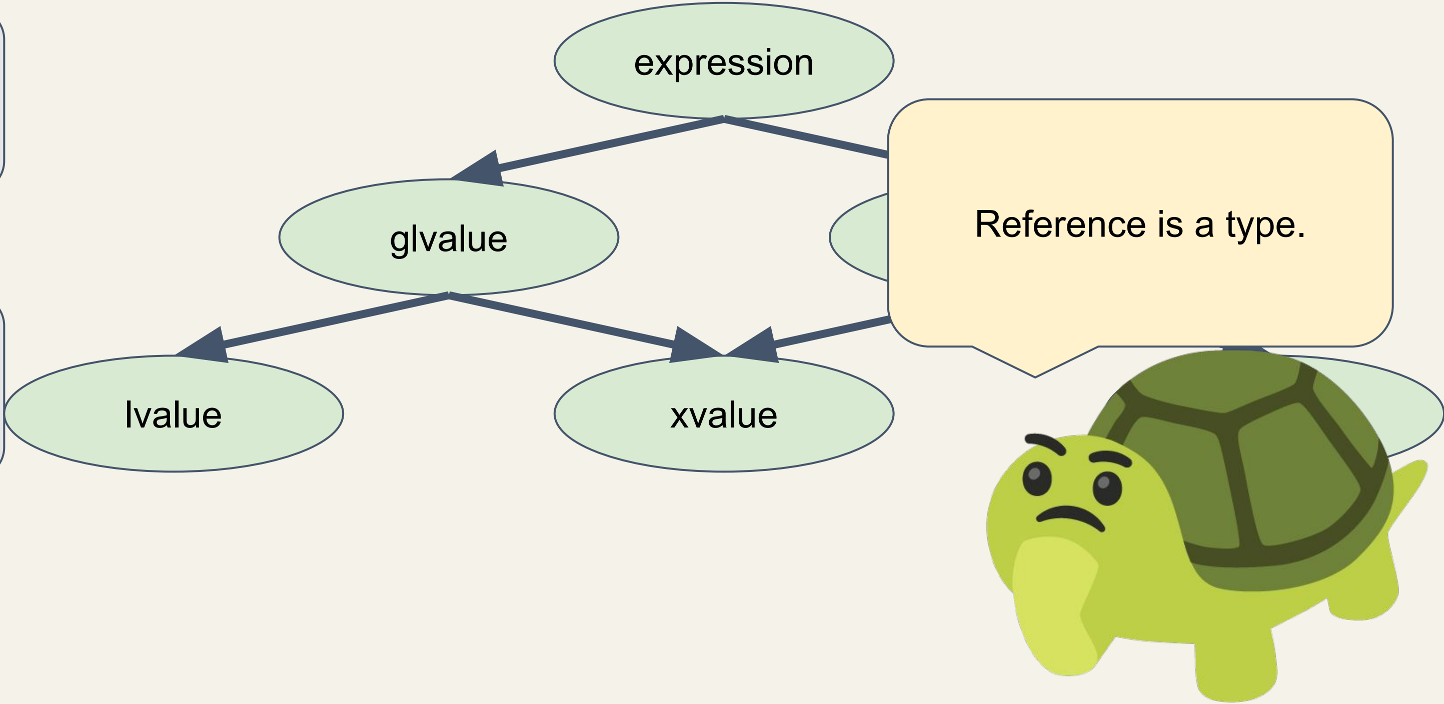


Reference

Declares a named variable as a reference, that is, an alias to an already-existing object or function.

Lvalue reference

Rvalue reference



Rules in templates

An *id-expression* naming a non-type *template-parameter* of class type T denotes a static storage duration object of type const T, known as a *template parameter object*, which is template-argument-equivalent ([temp.type]) to the corresponding template argument after it has been converted to the type of the *template-parameter* ([temp.arg.nontype]). No two template parameter objects are template-argument-equivalent.

[*Note 3*: If an *id-expression* names a non-type non-reference *template-parameter*, then it is a prvalue if it has non-class type. Otherwise, if it is of class type T, it is an lvalue and has type const T ([expr.prim.id.unqual]). — *end note*]

頭腦體操！

```
using X = int;
struct A {};
template<const X& x, int i, A a> void f() {
    i++;
    &x;

    &i;
    &a;
    int& ri = i;
    const int& cri = i;
    const A& ra = a;
}
```

頭腦體操！

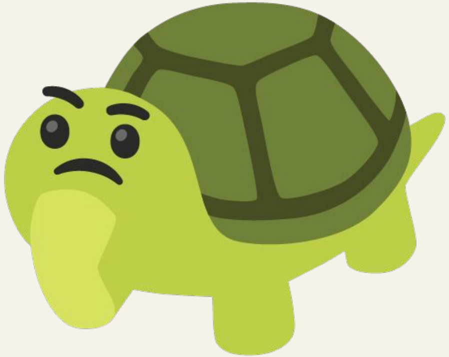
```
using X = int;
struct A {};
template<const X& x, int i, A a> void f() {
    i++;
    &x;

    &i;
    &a;
    int& ri = i;
    const int& cri = i;
    const A& ra = a;
}
```

頭腦體操！

```
using X = int;
struct A {};
template<const X& x, int i, A a> void f() {
    i++; // error: change of template-parameter value
    &x;

    &i;
    &a;
    int& ri = i;
    const int& cri = i;
    const A& ra = a;
}
```

頭腦體操！

```
using X = int;
struct A {};
template<const X& x, int i, A a> void f() {
    i++;

    &x;

    &i;
    &a;
    int& ri = i;
    const int& cri = i;
    const A& ra = a;
}
```



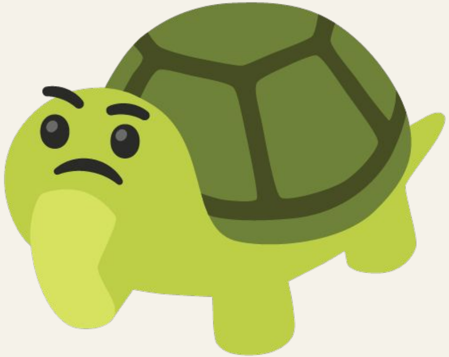
頭腦體操！

```
using X = int;
struct A {};
template<const X& x, int i, A a> void f() {
    i++;

    &x; // OK


    &i;
    &a;
    int& ri = i;
    const int& cri = i;
    const A& ra = a;
}
```

頭腦體操！

TEAMT5 杜浦數位安全

```
using X = int;
struct A {};
template<const X& x, int i, A a> void f() {
    i++;
    &x;

    &i;
    &a;
    int& ri = i;
    const int& cri = i;
    const A& ra = a;
}
```



頭腦體操！

```
using X = int;
struct A {};
template<const X& x, int i, A a> void f() {
    i++;
    &x;

    &i; // error: address of non-reference template-parameter
    &a;
    int& ri = i;
    const int& cri = i;
    const A& ra = a;
}
```

頭腦體操！

```
using X = int;
struct A {};
template<const X& x, int i, A a> void f() {
    i++;
    &x;

    &i;
    &a;
    int& ri = i;
    const int& cri = i;
    const A& ra = a;
}
```

頭腦體操！

```
using X = int;
struct A {};
template<const X& x, int i, A a> void f() {
    i++;
    &x;

    &i;
    &a;
    int& ri = i;
    const int& cri = i;
    const A& ra = a;
}
```