Contemporary C++: Learning Modern C++ in a Modern Way

الماس فناوری ابری پاسارگاد- آلفا

مدرس: سعید امراللهی بیوکی

Agenda 14/24

Session 14. Introduction to Templates (Part I)

- Why templates?
- Function templates
- Class templates
- Template terminologies
- Simple templates, simple effects
- Fundamental concepts: Instantiation, Specialization, Argument deduction, and Type deduction
- Q&A

150 min (incl. Q & A)

Template

Template

A *template* defines a family of classes, functions, or variables, an alias for a family of types, or a concept. from *Committee Draft, Chapter 13, section 13.1 [temp.pre]*

Template

> A *template* defines a family of classes, functions, or variables, an alias for a family of types, or a concept. from *Committee Draft, Chapter 13, section 13.1 [temp.pre]*

- Classes and functions: C++98
- alias for a family of types: C++11
- variables: C++14
- concepts: C++20

Template

A *template* defines a family of classes, functions, or variables, an alias for a family of types, or a concept. from *Committee Draft, Chapter 13, section 13.1 [temp.pre]*

- Classes and functions: C++98
- alias for a family of types: C++11
- variables: C++14
- concepts: C++20

- When you write a template you are writing the specification for an infinite set of classes (for class templates) or functions (for template functions).

  - Rob Murray, C++ Strategies and Tactics, 1993

why Templates?

why Templates?

1. You can implement the same behavior again and again for each type that needs this behavior.

→ ARM C++, Java, C#.

- ARM C++: The Annotated C++ Reference Manual A.K.A your father's C++ ☺

- Reinvent the wheel

why Templates?

1. You can implement the same behavior again and again for each type that needs this behavior.
→ ARM C++, Java, C#.

- ARM C++: The Annotated C++ Reference Manual A.K.A your father's C++ ☺

- Reinvent the wheel

2. You can write general code for a common base type such as Object or void*. → ARM C++, Java, C#.

why Templates?

1. You can implement the same behavior again and again for each type that needs this behavior.
→ ARM C++, Java, C#.

- ARM C++: The Annotated C++ Reference Manual A.K.A your father's C++ ☺

- Reinvent the wheel

2. You can write general code for a common base type such as Object or void*. → ARM C++, Java, C#.

- There is no generic/universal root class like Object!

why Templates?

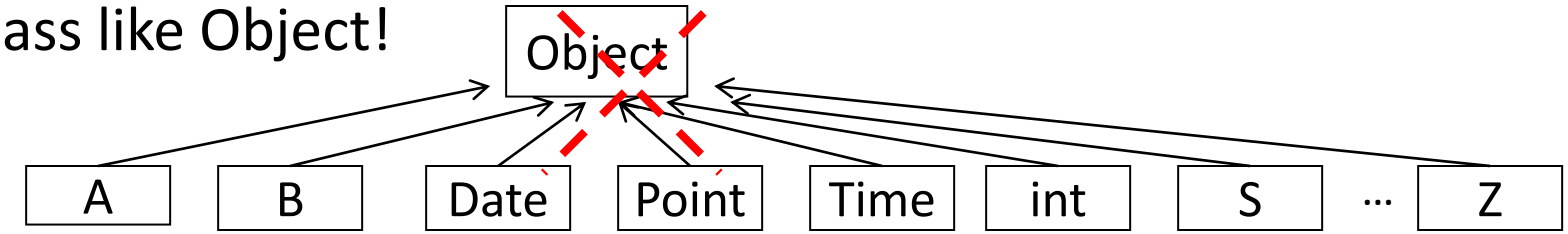
1. You can implement the same behavior again and again for each type that needs this behavior.
→ ARM C++, Java, C#.

- ARM C++: The Annotated C++ Reference Manual A.K.A your father's C++ ☺

- Reinvent the wheel

2. You can write general code for a common base type such as Object or void*. → ARM C++, Java, C#.

- There is no generic/universal root class like Object!



why Templates?

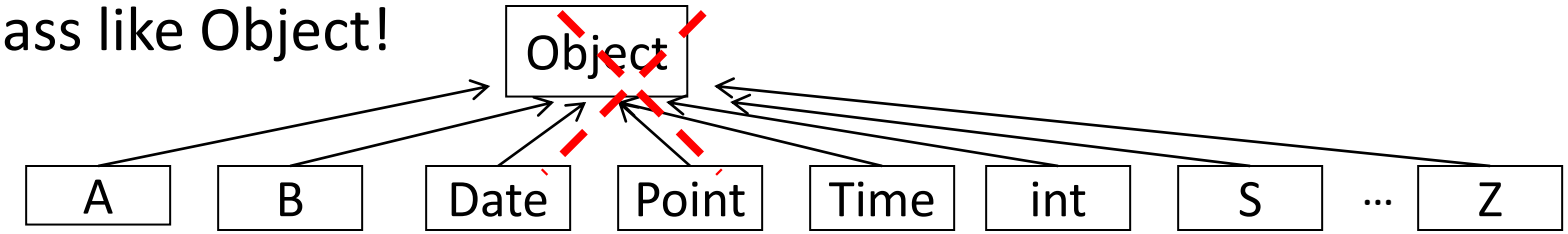
1. You can implement the same behavior again and again for each type that needs this behavior.
→ ARM C++, Java, C#.

- ARM C++: The Annotated C++ Reference Manual A.K.A your father's C++ ☺

- Reinvent the wheel

2. You can write general code for a common base type such as Object or void*. → ARM C++, Java, C#.

- There is no generic/universal root class like Object!
  - No semantics
  - Sloppy interfaces
  - You lose the benefit of type-checking.



why Templates?

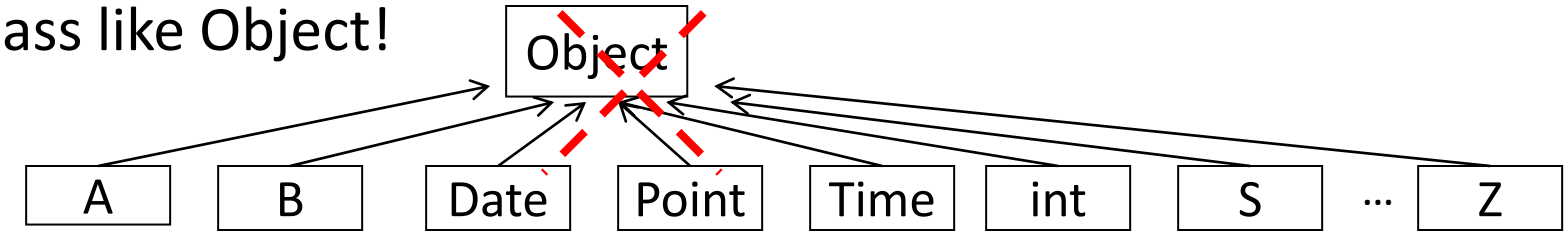
1. You can implement the same behavior again and again for each type that needs this behavior.
→ ARM C++, Java, C#.

- ARM C++: The Annotated C++ Reference Manual A.K.A your father's C++ ☺

- Reinvent the wheel

2. You can write general code for a common base type such as Object or void*. → ARM C++, Java, C#.

- There is no generic/universal root class like Object!
  - No semantics
  - Sloppy interfaces
  - You lose the benefit of type-checking.



3. You can use a special preprocessor. → C

The drawbacks of preprocessor: stupid text replacement mechanisms.

why Templates?

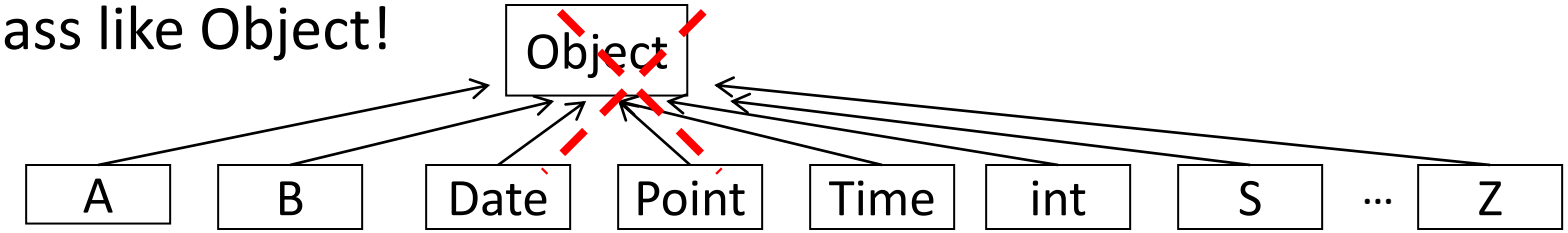
1. You can implement the same behavior again and again for each type that needs this behavior.
→ ARM C++, Java, C#.

- ARM C++: The Annotated C++ Reference Manual A.K.A your father's C++ ☺

- Reinvent the wheel

2. You can write general code for a common base type such as Object or void*. → ARM C++, Java, C#.

- There is no generic/universal root class like Object!
  - No semantics
  - Sloppy interfaces
  - You lose the benefit of type-checking.



3. You can use a special preprocessor. → C

The drawbacks of preprocessor: stupid text replacement mechanisms.

```
#define MAX(A, B) ((A) > (B) ? (A) : (B))
```

```
int a = 1; b = 0;
MAX(a++, b);
```

why Templates?

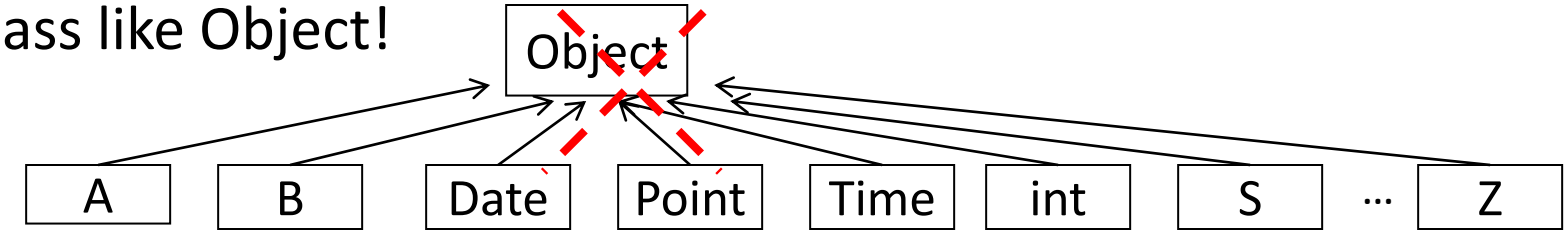
1. You can implement the same behavior again and again for each type that needs this behavior.
→ ARM C++, Java, C#.

- ARM C++: The Annotated C++ Reference Manual A.K.A your father's C++ ☺

- Reinvent the wheel

2. You can write general code for a common base type such as Object or void*. → ARM C++, Java, C#.

- There is no generic/universal root class like Object!
  - No semantics
  - Sloppy interfaces
  - You lose the benefit of type-checking.



3. You can use a special preprocessor. → C

The drawbacks of preprocessor: stupid text replacement mechanisms.

```
#define MAX(A, B) ((A) > (B) ? (A) : (B))
```

```
int a = 1; b = 0;
MAX(a++, b);
```

- Support vs. Enable


Max Macros
Prog.

Function template

Function template

- A function template defines a family of functions.

Function template

- A function template defines a family of functions.
- Example: The absolute function

1
```
int abs(int a)
{
    return a >= 0 ? a : -a;
}
```

2
```
double abs(double a)
{
    return a >= 0 ? a : -a;
}
```

3
```
float abs(float a)
{
    return a >= 0 ? a : -a;
}
```

4
```
int abs(long long a)
{
    return a >= 0 ? a : -a;
}
```

5
```
double abs(char a)
{
    return a >= 0 ? a : -a;
}
```

...

Function template

Reinvent the wheel!!

- A function template defines a family of functions.
- Example: The absolute function

1
```
int abs(int a)
{
    return a >= 0 ? a : -a;
}
```

2
```
double abs(double a)
{
    return a >= 0 ? a : -a;
}
```

3
```
float abs(float a)
{
    return a >= 0 ? a : -a;
}
```

4
```
int abs(long long a)
{
    return a >= 0 ? a : -a;
}
```

5
```
double abs(char a)
{
    return a >= 0 ? a : -a;
}
```

...

Function template

- A function template defines a family of functions.
- Example: The absolute function

Reinvent the wheel!!

1
```
int abs(int a)
{
    return a >= 0 ? a : -a;
}
```

2
```
double abs(double a)
{
    return a >= 0 ? a : -a;
}
```

3
```
float abs(float a)
{
    return a >= 0 ? a : -a;
}
```

4
```
int abs(long long a)
{
    return a >= 0 ? a : -a;
}
```

5
```
double abs(char a)
{
    return a >= 0 ? a : -a;
}
```

...

```
template<class T>
T abs(T a)
{
    return a >= 0 ? a : -a;
}
```

Function template

Reinvent the wheel!!

- A function template defines a family of functions.
- Example: The absolute function

1
```
int abs(int a)
{
    return a >= 0 ? a : -a;
}
```

2
```
double abs(double a)
{
    return a >= 0 ? a : -a;
}
```

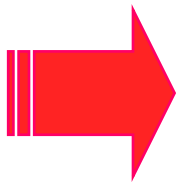
3
```
float abs(float a)
{
    return a >= 0 ? a : -a;
}
```

4
```
int abs(long long a)
{
    return a >= 0 ? a : -a;
}
```

5
```
double abs(char a)
{
    return a >= 0 ? a : -a;
}
```

...

```
template<class T>
T abs(T a)
{
    return a >= 0 ? a : -a;
}
```



```
template<class T>
T abs(const T& a)
{
    return a >= 0 ? a : -a;
}
```

- Avoid expensive and redundant copy

Function template

Reinvent the wheel!!

- A function template defines a family of functions.
- Example: The absolute function

1
```
int abs(int a)
{
    return a >= 0 ? a : -a;
}
```

2
```
double abs(double a)
{
    return a >= 0 ? a : -a;
}
```

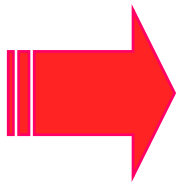
3
```
float abs(float a)
{
    return a >= 0 ? a : -a;
}
```

4
```
int abs(long long a)
{
    return a >= 0 ? a : -a;
}
```

5
```
double abs(char a)
{
    return a >= 0 ? a : -a;
}
```

...

```
template<class T>
T abs(T a)
{
    return a >= 0 ? a : -a;
}
```



```
template<class T>
T abs(const T& a)
{
    return a >= 0 ? a : -a;
}
```

- Avoid expensive and redundant copy

- Very simple lifting

Absolute function- more details

Absolute function- more details

- The power of C++ function name overloading.

Absolute function- more details

- The power of C++ function name overloading.
- The abs family functions in C programming language

```
int         abs(int);
long        labs( long n );
long long   llabs( long long n );
intmax_t    imaxabs( intmax_t n );
float       fabsf( float arg );
double      fabs( double arg );
long double fabsl( long double arg );
long double fabsl( long double arg );
double      cabs( double complex z );
long double cabsl( long double complex z);
```

Absolute function- more details

- The power of C++ function name overloading.
- The abs family functions in C programming language

- Type T requirement:
  - GreaterThanOrEqualTo comparable
  - Unary minus operator
  - CopyConstructible

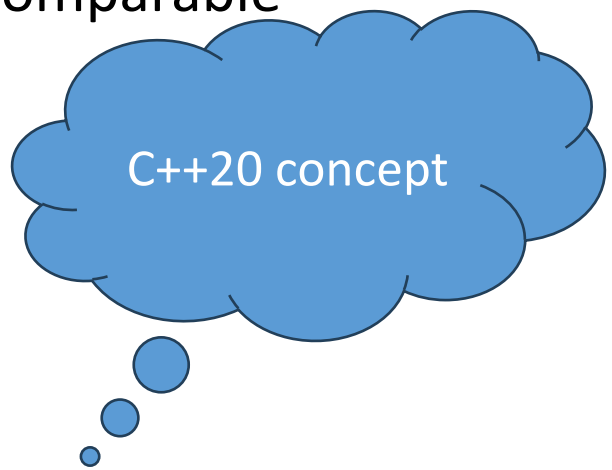
```
int         abs(int);
long        labs( long n );
long long   llabs( long long n );
intmax_t    imaxabs( intmax_t n );
float       fabsf( float arg );
double      fabs( double arg );
long double fabsl( long double arg );
long double fabsl( long double arg );
double      cabs( double complex z );
long double cabsl( long double complex z);
```

Absolute function- more details

- The power of C++ function name overloading.
- The abs family functions in C programming language

```
int          abs(int);
long         labs( long n );
long long    llabs( long long n );
intmax_t     imaxabs( intmax_t n );
float        fabsf( float arg );
double       fabs( double arg );
long double  fabsl( long double arg );
long double  fabsl( long double arg );
double       cabs( double complex z );
long double  cabsl( long double complex z);
```

- Type T requirement:
  - GreaterThanOrEqualTo comparable
  - Unary minus operator
  - CopyConstructible

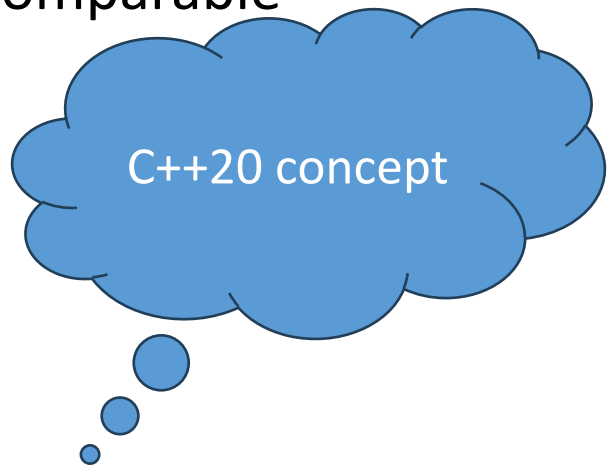


Absolute function- more details

- The power of C++ function name overloading.
- The abs family functions in C programming language

```
int         abs(int);
long        labs( long n );
long long   llabs( long long n );
intmax_t    imaxabs( intmax_t n );
float       fabsf( float arg );
double      fabs( double arg );
long double fabsl( long double arg );
long double fabsl( long double arg );
double      cabs( double complex z );
long double cabsl( long double complex z);
```

- Type T requirement:
  - GreaterThanOrEqualTo comparable
  - Unary minus operator
  - CopyConstructible



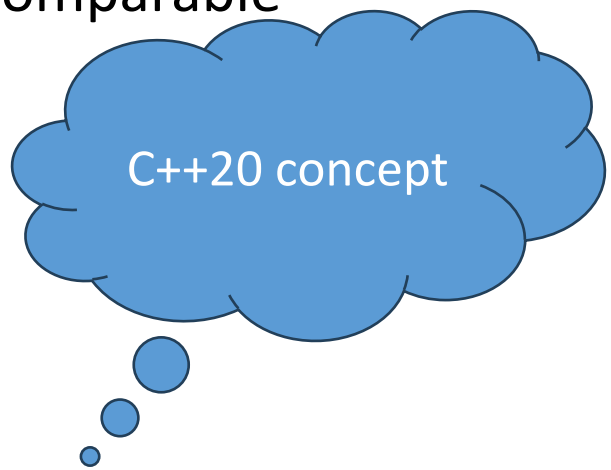
- T is open for family of types.
- T, Type, …

Absolute function- more details

- The power of C++ function name overloading.
- The abs family functions in C programming language

```
int          abs(int);
long         labs( long n );
long long    llabs( long long n );
intmax_t     imaxabs( intmax_t n );
float        fabsf( float arg );
double       fabs( double arg );
long double  fabsl( long double arg );
long double  fabsl( long double arg );
double       cabs( double complex z );
long double  cabsl( long double complex z);
```

- Type T requirement:
  - GreaterThanOrEqualTo comparable
  - Unary minus operator
  - CopyConstructible



- T is open for family of types.
- T, Type, …

```
template<class T>
T abs(const T& a)
{
    return a >= 0 ? a : -a;
}
```

Absolute function- user-defined types

1

```
class X { // an int wrapper
    friend bool operator>=(const X&, const X&);
    friend X operator-(const X&);
    int i;
public:
    X(int ii = 0) : i{ ii } {} // default ctor
};
```

2

```
namespace { // unnamed namespace
    template<class T>
    T abs(const T& t)
    {
        return t >= 0 ? t : -t;
    }
}
```

3

```
void f()
{
    X positive = abs(X{-1});
}
```

AbsFuncTypeReqTest Prog.

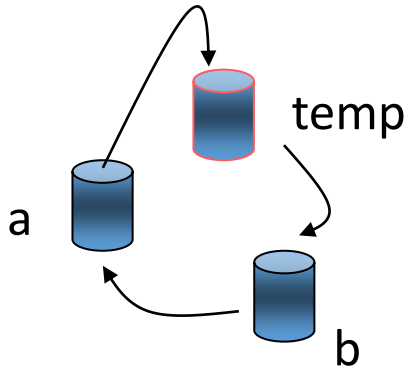
Function template- swap

Function template- swap

- Copy-based swap rather than Move-based one!

Function template- swap

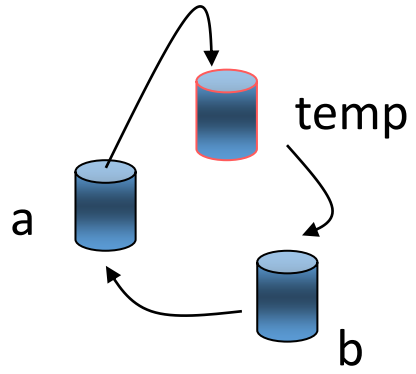
- Copy-based swap rather than Move-based one!




```
void swap(int& a, int& b)
{
  const int temp = a;
  a = b;
  b = temp;
}
// …
int a = 10, b = 20;
swap(a, b); // now b = 10, a = 20
```

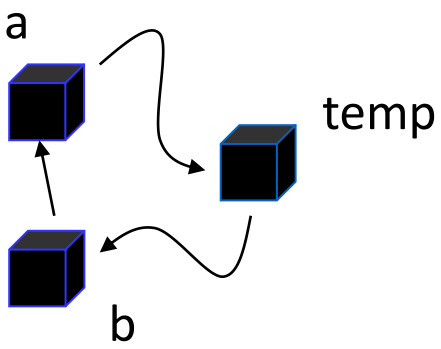
Function template- swap

- Copy-based swap rather than Move-based one!



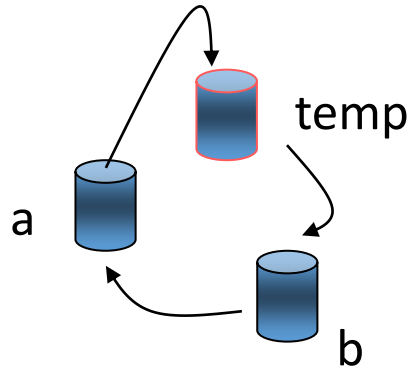

```
void swap(int& a, int& b)
{
  const int temp = a;
  a = b;
  b = temp;
}
// …
int a = 10, b = 20;
swap(a, b); // now b = 10, a = 20
```

```
void swap(double& a, double& b)
{
  const double temp = a;
  a = b;
  b = temp;
}
// …
double d1(3.14), d2(2.72);
swap(d1, d2); // now b = 3.14, a = 2.72
```



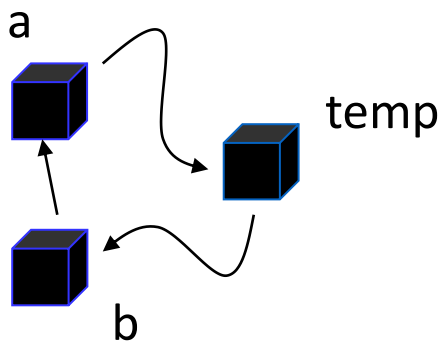
Function template- swap

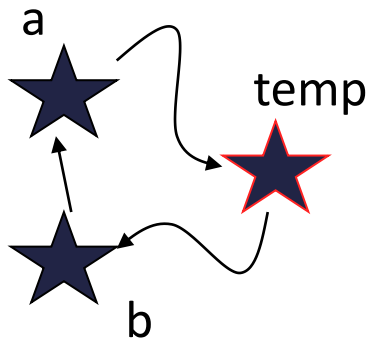
- Copy-based swap rather than Move-based one!




```
void swap(int& a, int& b)
{
  const int temp = a;
  a = b;
  b = temp;
}
// …
int a = 10, b = 20;
swap(a, b); // now b = 10, a = 20
```

```
void swap(double& a, double& b)
{
  const double temp = a;
  a = b;
  b = temp;
}
// …
double d1(3.14), d2(2.72);
swap(d1, d2); // now b = 3.14, a = 2.72
```

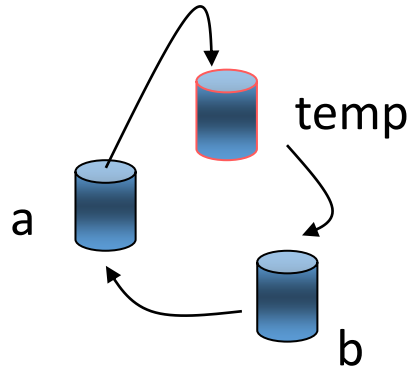





```
void swap(std::string& a, std::string& b)
{
  const std::string temp = a;
  a = b;
  b = temp;
}
// …
std::string Hi("Hello"), Bye("Bye");
swap(a, b); // now Hi = "Bye", Bye = "Hello"
```

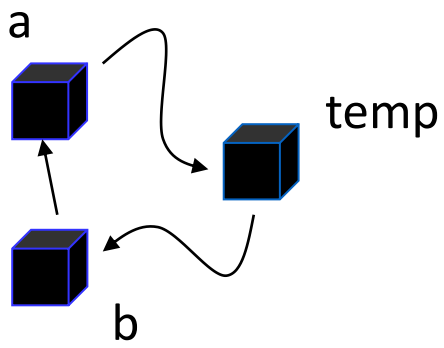
Function template- swap

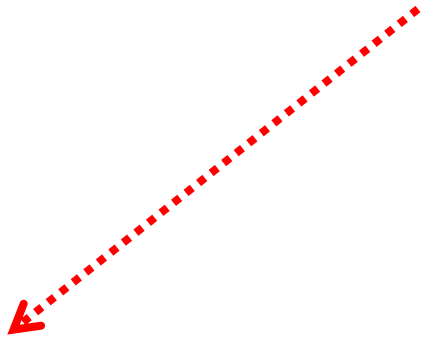
- Copy-based swap rather than Move-based one!



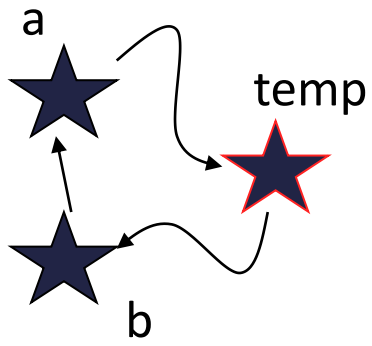

```
void swap(int& a, int& b)
{
  const int temp = a;
  a = b;
  b = temp;
}
// …
int a = 10, b = 20;
swap(a, b); // now b = 10, a = 20
```

```
void swap(double& a, double& b)
{
  const double temp = a;
  a = b;
  b = temp;
}
// …
double d1(3.14), d2(2.72);
swap(d1, d2); // now b = 3.14, a = 2.72
```





Reinvent the wheel!!




```
void swap(std::string& a, std::string& b)
{
  const std::string temp = a;
  a = b;
  b = temp;
}
// …
std::string Hi("Hello"), Bye("Bye");
swap(a, b); // now Hi = "Bye", Bye = "Hello"
```

Function template- swap cont.

Function template- swap cont.

```
template<class T>
void Swap(T& a, T& b) // family of swaps
{
  const T temp = a; // copy ctor
  a = b; // assignment operator
  b = temp;
}
```

Function template- swap cont.

```
template<class T>
void Swap(T& a, T& b) // family of swaps
{
  const T temp = a; // copy ctor
  a = b; // assignment operator
  b = temp;
}
```

```
// use Swap
void f()
{
  int i = 2, j = 3;
  Swap(i, j); // swap two ints

  bool b1 = false, b2 = true;
  Swap(b1, b2); // swap two bools

  std::string Hi("Hi"), Bye("Bye");
  Swap(Hi, Bye); // swap two strings
}
```

Function template- swap cont.

```
template<class T>
void Swap(T& a, T& b) // family of swaps
{
  const T temp = a; // copy ctor
  a = b; // assignment operator
  b = temp;
}
```

```
// use Swap
void f()
{
  int i = 2, j = 3;
  Swap(i, j); // swap two ints

  bool b1 = false, b2 = true;
  Swap(b1, b2); // swap two bools

  std::string Hi("Hi"), Bye("Bye");
  Swap(Hi, Bye); // swap two strings
}
```

- For calling Swap on a type as a template argument it must have copy constructor and assignment operator. In other words, it should be *Copy constructible* and *Assignable*.

Function template- swap cont.

```
template<class T>
void Swap(T& a, T& b) // family of swaps
{
  const T temp = a; // copy ctor
  a = b; // assignment operator
  b = temp;
}
```

```
// use Swap
void f()
{
  int i = 2, j = 3;
  Swap(i, j); // swap two ints

  bool b1 = false, b2 = true;
  Swap(b1, b2); // swap two bools

  std::string Hi("Hi"), Bye("Bye");
  Swap(Hi, Bye); // swap two strings
}
```

- For calling Swap on a type as a template argument it must have copy constructor and assignment operator. In other words, it should be *Copy constructible* and *Assignable*.
- Another implementation

Function template- swap cont.

```
template<class T>
void Swap(T& a, T& b) // family of swaps
{
  const T temp = a; // copy ctor
  a = b; // assignment operator
  b = temp;
}
```

```
// use Swap
void f()
{
  int i = 2, j = 3;
  Swap(i, j); // swap two ints

  bool b1 = false, b2 = true;
  Swap(b1, b2); // swap two bools

  std::string Hi("Hi"), Bye("Bye");
  Swap(Hi, Bye); // swap two strings
}
```

- For calling Swap on a type as a template argument it must have copy constructor and assignment operator. In other words, it should be *Copy constructible* and *Assignable*.
- Another implementation

```
template<class T>
void Swap(T& a, T& b) // family of swaps
{
  T temp; // default ctor
  temp = a; // assignment operator
  a = b; // assignment operator
  b = temp;
}
```

Function template- swap cont.

```
template<class T>
void Swap(T& a, T& b) // family of swaps
{
  const T temp = a; // copy ctor
  a = b; // assignment operator
  b = temp;
}
```

```
// use Swap
void f()
{
  int i = 2, j = 3;
  Swap(i, j); // swap two ints

  bool b1 = false, b2 = true;
  Swap(b1, b2); // swap two bools

  std::string Hi("Hi"), Bye("Bye");
  Swap(Hi, Bye); // swap two strings
}
```

- For calling Swap on a type as a template argument it must have copy constructor and assignment operator. In other words, it should be *Copy constructible* and *Assignable*.
- Another implementation

```
template<class T>
void Swap(T& a, T& b) // family of swaps
{
  T temp; // default ctor
  temp = a; // assignment operator
  a = b; // assignment operator
  b = temp;
}
```

Default constructible

Assignable

Typename

Typename

- class vs. typename
- There is no semantic difference between *class* and *typename* in a template parameter.

Typename

- class vs. typename
- There is no semantic difference between *class* and *typename* in a template parameter.

```
template<class T>
T abs(const T& a)
{
    return a >= 0 ? a : -a;
}
```

VS.

```
template<typename T>
T abs(const T& a)
{
    return a >= 0 ? a : -a;
}
```

Typename

- class vs. typename
- There is no semantic difference between *class* and *typename* in a template-parameter.

```
template<class T>
T abs(const T& a)
{
    return a >= 0 ? a : -a;
}
```

vs.

```
template<typename T>
T abs(const T& a)
{
    return a >= 0 ? a : -a;
}
```

- Any types: Fundamental types, user-defined types, …
- class type: class, struct, union

Typename

- class vs. typename
- There is no semantic difference between *class* and *typename* in a template-parameter.

```
template<class T>
T abs(const T& a)
{
    return a >= 0 ? a : -a;
}
```

VS.

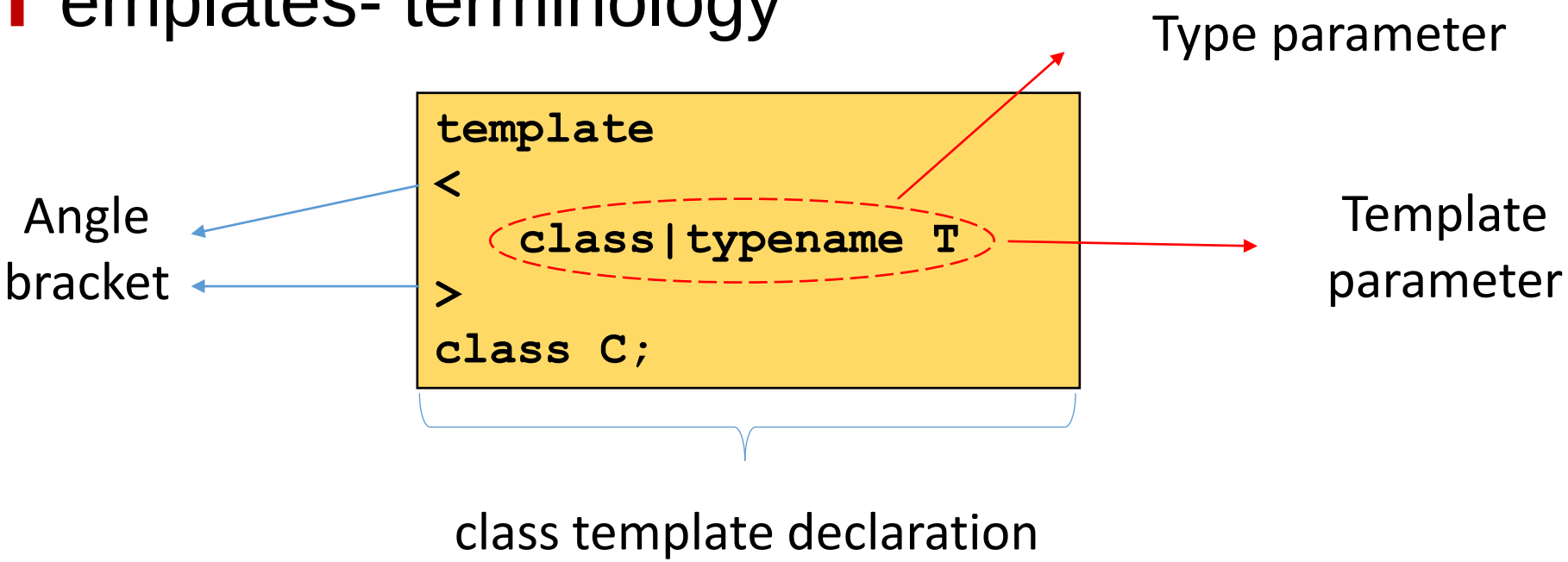
```
template<typename T>
T abs(const T& a)
{
    return a >= 0 ? a : -a;
}
```

- Any types: Fundamental types, user-defined types, …
- class type: class, struct, union
- The keyword struct or union can not be used in place of the keyword typename.

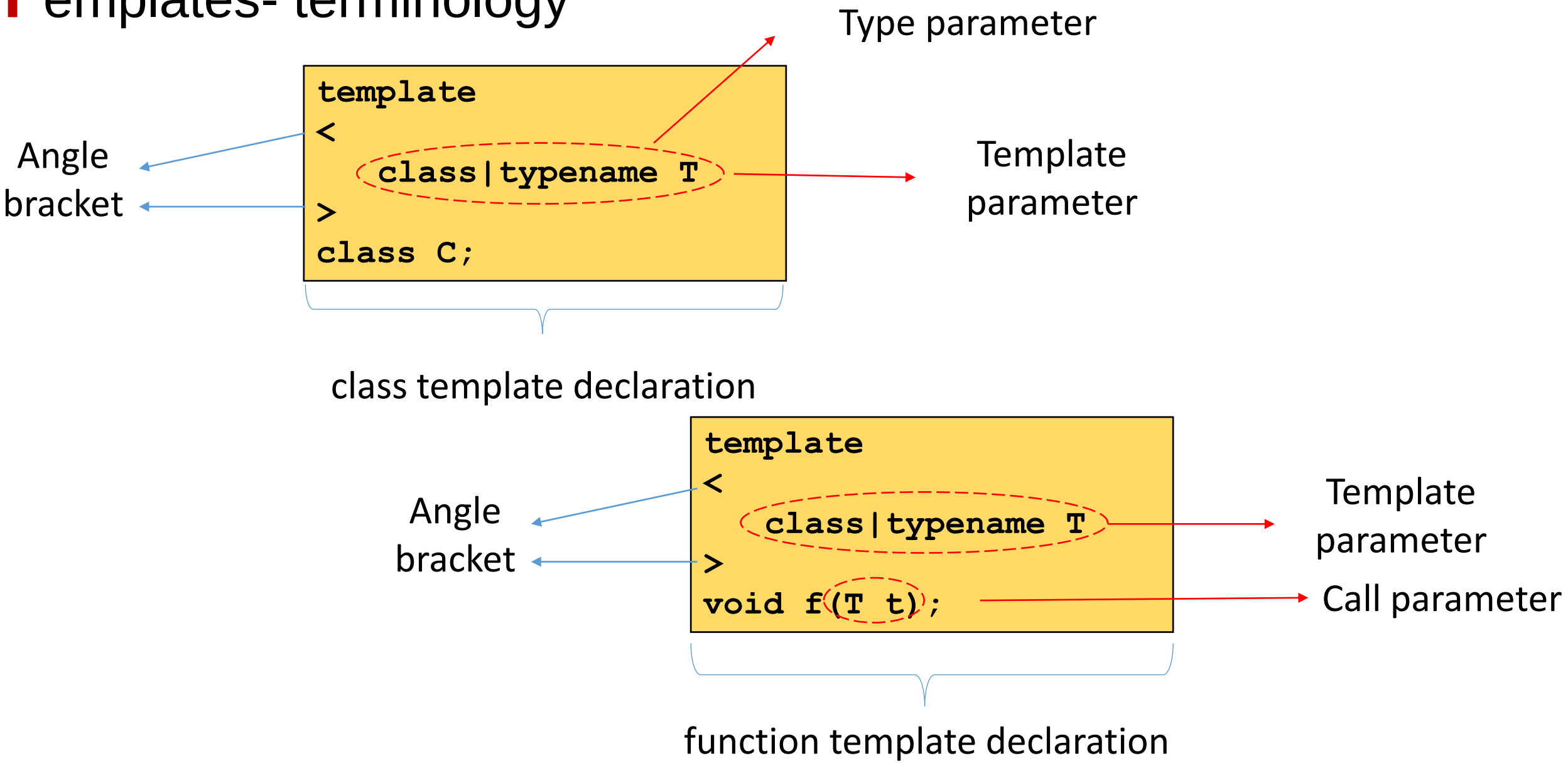
```
template<class T> T sqrt(T); // OK
template<typename T> T sqrt(T); // OK
template<struct T> T sqrt(T); // error
template<union T> T sqrt(T); // error
```

Templates- terminology

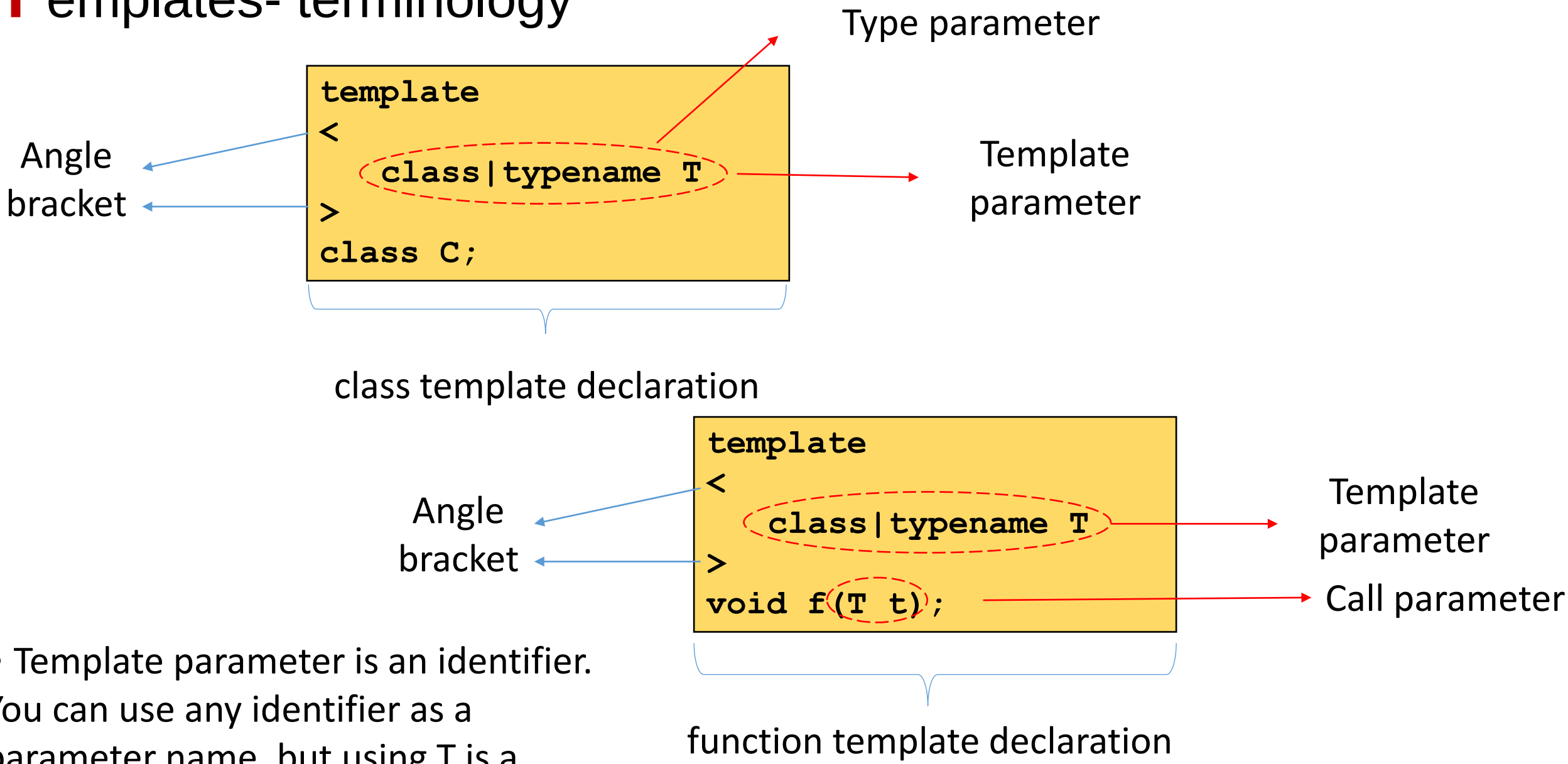
Templates- terminology



Templates- terminology



Templates- terminology



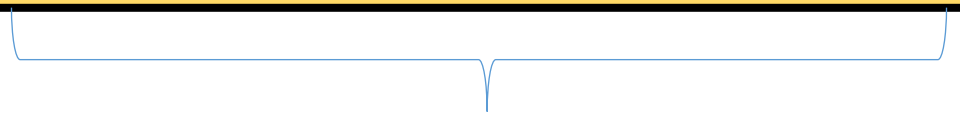
- Template parameter is an identifier.

You can use any identifier as a parameter name, but using T is a convention.

Templates- terminology cont.

Templates- terminology cont.

```
template<class T1, class T2, class T3> class C;
```



Template parameter list

Templates- declaration and definition

Templates- declaration and definition

- Inside template classes and functions template parameters can be used just like any other type to declare local variables, data members, and member functions.

Templates- declaration and definition

- Inside template classes and functions template parameters can be used just like any other type to declare local variables, data members, and member functions.
- Members of a template class are themselves templates parameterized by the parameters of their template class. When such a member is defined outside its class, it must explicitly be declared a template.

Templates- declaration and definition

- Inside template classes and functions template parameters can be used just like any other type to declare local variables, data members, and member functions.
- Members of a template class are themselves templates parameterized by the parameters of their template class. When such a member is defined outside its class, it must explicitly be declared a template.

1

```
template<class T>
class Stack {
  T* v;
  int top;
  int size;
public:
  Stack(int = 10);
  bool is_empty();
  bool is_full();
  void push(T);
  T pop();
  ~Stack();
};
```

Templates- declaration and definition

- Inside template classes and functions template parameters can be used just like any other type to declare local variables, data members, and member functions.
- Members of a template class are themselves templates parameterized by the parameters of their template class. When such a member is defined outside its class, it must explicitly be declared a template.

1

```
template<class T>
class Stack {
  T* v;
  int top;
  int size;
public:
  Stack(int = 10);
  bool is_empty();
  bool is_full();
  void push(T);
  T pop();
  ~Stack();
};
```

2

```
template<class T>
Stack<T>::Stack(int s) : v(new T[size = s]), top(-1) {}
template<class T>
inline bool Stack<T>::IsEmpty() { /* … */ }
template<class T>
inline bool Stack<T>::IsFull() { /* … */ }
/*
  …
*/
```

Templates- declaration and definition

- Inside template classes and functions template parameters can be used just like any other type to declare local variables, data members, and member functions.
- Members of a template class are themselves templates parameterized by the parameters of their template class. When such a member is defined outside its class, it must explicitly be declared a template.

1

```
template<class T>
class Stack {
  T* v;
  int top;
  int size;
public:
  Stack(int = 10);
  bool is_empty();
  bool is_full();
  void push(T);
  T pop();
  ~Stack();
};
```

2

```
template<class T>
Stack<T>::Stack(int s) : v(new T[size = s]), top(-1) {}
template<class T>
inline bool Stack<T>::IsEmpty() { /* … */ }
template<class T>
inline bool Stack<T>::IsFull() { /* … */ }
/*
  …
*/
```

- The name of class template can't be overloaded:

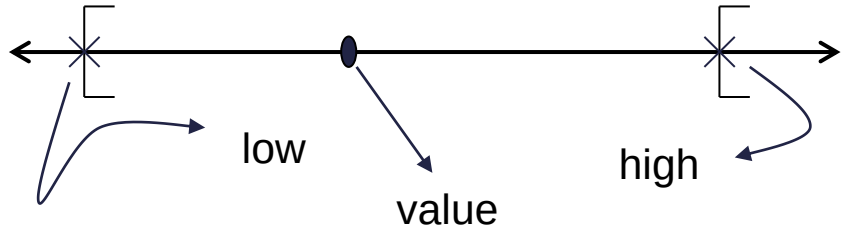
```
template<class T>
class X { /*… */ };

class X { /* … */ }; // error: double definition
```

(Non-template) Range concept

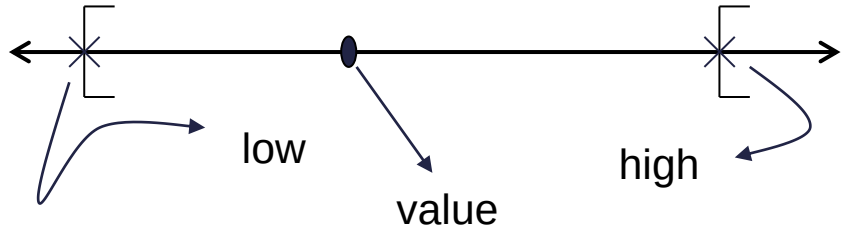
(Non-template) Range concept

- Half-Open range: [low, high)



(Non-template) Range concept

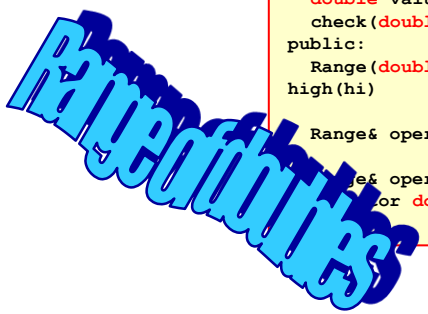
- Half-Open range: [low, high)




```
// Range concept  (for integers)
class Range { // simple value type
  int value, low, high;  // invariant: low <= value < high
  check(int v) { if (!(low<=v && v<high)) throw Range_error(); }
public:
  Range(int lo, int v, int hi) : low(lo), value(v), high(hi) { check(v); }
  Range& operator=(const Range& a) { check(a.v); value = a.value;
        return *this; }
  Range& operator=(int a) { check(a); value = a; return *this; }
  operator int() { return value; }
};
```

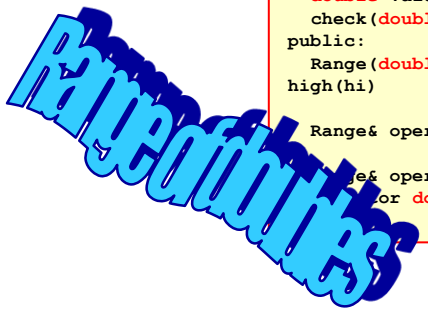
Range concept- Template version

Range concept- Template version

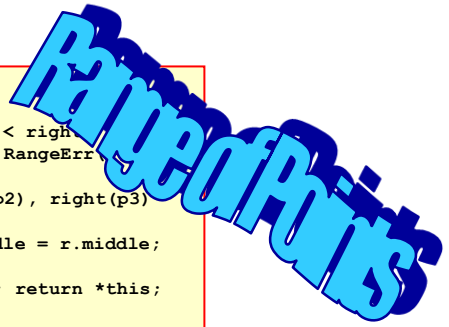



```
// Range concept (for doubles)
class Range { // simple value type
  double value, low, high;  // invariant: low <= value < high
  check(double v) { if (!(low<=v && v<high)) throw Range_error(); }
public:
  Range(double lo, double v, double hi) : low(lo), value(v),
high(hi)
                { check(v); }
  Range& operator=(const Range& a) { check(a.v); value = a.value;
                return *this; }
  Range& operator=(double a) { check(a); value = a; return *this; }
  operator double() { return value; }
```

Range concept- Template version

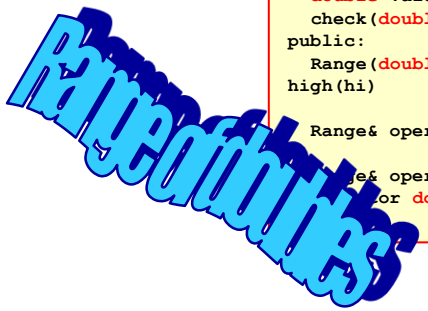



```
// Range concept (for doubles)
class Range { // simple value type
  double value, low, high;  // invariant: low <= value < high
  check(double v) { if (!(low<=v && v<high)) throw Range_error(); }
public:
  Range(double lo, double v, double hi) : low(lo), value(v),
high(hi)
                { check(v); }
  Range& operator=(const Range& a) { check(a.v); value = a.value;
                return *this; }
  Range& operator=(double a) { check(a); value = a; return *this; }
  operator double() { return value; }
```



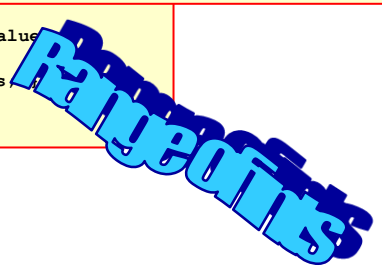

```
// Range concept (for points)
class Range { // Simple value type
  Point left, middle, right; // Invariant: left <= middle < righ
  void check(Point p) { if (left > p || p >= right) throw RangeErr
public:
  Range(Point p1, Point p2, Point p3) : left(p1), middle(p2), right(p3)
                { check(middle); }
  Range& operator=(const Range& r) { check(r.middle); middle = r.middle;
                return *this; }
  Range& operator=(const Point& p) { check(p); middle = p; return *this;
}
  operator Point() const { return middle; }
};
```

Range concept- Template version

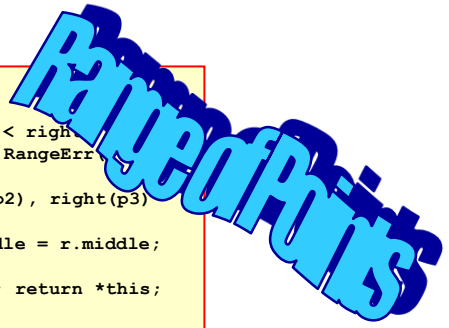



```
// Range concept (for doubles)
class Range { // simple value type
  double value, low, high;  // invariant: low <= value < high
  check(double v) { if (!(low<=v && v<high)) throw Range_error(); }
public:
  Range(double lo, double v, double hi) : low(lo), value(v),
high(hi)
                { check(v); }
  Range& operator=(const Range& a) { check(a.v); value = a.value;
                ret
  ge& operator=(d
  or double()
```

```
// Range concept  (for integers)
class Range { // simple value type
  int value, low, high;  // invariant: low <= value < h
  check(int v) { if (!(low<=v && v<high)) throw Range_e
public:
  Range(int lo, int v, int hi) : low(lo), value(v), hi
}
  Range& operator=(const Range& a) { check(a.v); value = a.value
                return *this; }
  Range& operator=(int a) { check(a); value = a; return *this;
  operator int() { return value; }
};
```




```
// Range concept (for points)
class Range { // Simple value type
  Point left, middle, right; // Invariant: left <= middle < righ
  void check(Point p) { if (left > p || p >= right) throw RangeErr
public:
  Range(Point p1, Point p2, Point p3) : left(p1), middle(p2), right(p3)
                { check(middle); }
  Range& operator=(const Range& r) { check(r.middle); middle = r.middle;
                return *this; }
  Range& operator=(const Point& p) { check(p); middle = p; return *this;
}
  operator Point() const { return middle; }
};
```

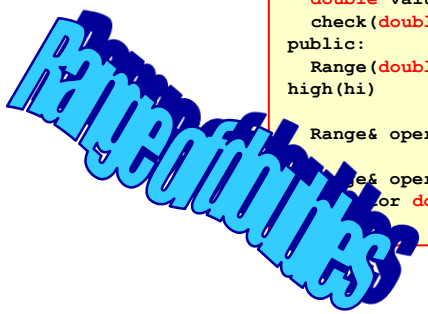


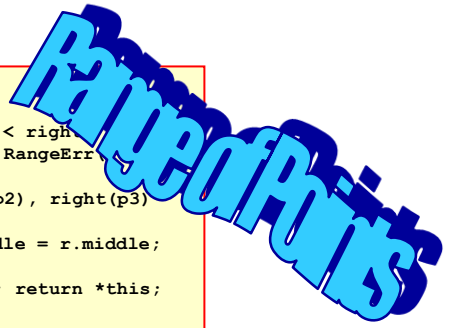
Range concept- Template version

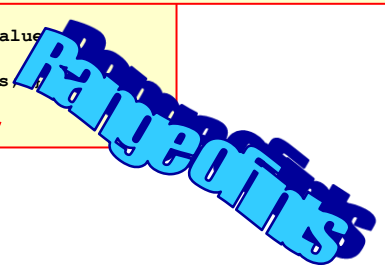
```
// Range concept (for doubles)
class Range { // simple value type
  double value, low, high;  // invariant: low <= value < high
  check(double v) { if (!(low<=v && v<high)) throw Range_error(); }
public:
  Range(double lo, double v, double hi) : low(lo), value(v),
high(hi)
                { check(v); }
  Range& operator=(const Range& a) { check(a.v); value = a.value;
```

```
// Range concept  (for integers)
class Range { // simple value type
  int value, low, high;  // invariant: low <= value <
  check(int v) { if (!(low<=v && v<high)) throw Range_
public:
  Range(int lo, int v, int hi) : low(lo), value(v), hi
}
  Range& operator=(const Range& a) { check(a.v); value = a.value
                return *this; }
  Range& operator=(int a) { check(a); value = a; return *this;
  operator int() { return value; }
};
```

```
// Range concept (for points)
class Range { // Simple value type
  Point left, middle, right; // Invariant: left <= middle < righ
  void check(Point p) { if (left > p || p >= right) throw RangeErr
public:
  Range(Point p1, Point p2, Point p3) : left(p1), middle(p2), right(p3)
                { check(middle); }
  Range& operator=(const Range& r) { check(r.middle); middle = r.middle;
                return *this; }
  Range& operator=(const Point& p) { check(p); middle = p; return *this;
}
  operator Point() const { return middle; }
};
```







Reinvent the wheel!!

Range concept- Template version

```
// Range concept (for doubles)
class Range { // simple value type
  double value, low, high;  // invariant: low <= value < high
  check(double v) { if (!(low<=v && v<high)) throw Range_error(); }
public:
  Range(double lo, double v, double hi) : low(lo), value(v),
high(hi)
                { check(v); }
 Range& operator=(const Range& a) { check(a.v); value = a.value;
```

```
// Range concept  (for integers)
class Range { // simple value type
  int value, low, high;  // invariant: low <= value <
  check(int v) { if (!(low<=v && v<high)) throw Range_
public:
  Range(int lo, int v, int hi) : low(lo), value(v), hi
}
  Range& operator=(const Range& a) { check(a.v); value = a.value
                return *this; }
  Range& operator=(int a) { check(a); value = a; return *this;
  operator int() { return value; }
};
```

```
// Range concept (for points)
class Range { // Simple value type
  Point left, middle, right; // Invariant: left <= middle < righ
  void check(Point p) { if (left > p || p >= right) throw RangeErr
public:
  Range(Point p1, Point p2, Point p3) : left(p1), middle(p2), right(p3)
                { check(middle); }
  Range& operator=(const Range& r) { check(r.middle); middle = r.middle;
                return *this; }
  Range& operator=(const Point& p) { check(p); middle = p; return *this;
}
  operator Point() const { return middle; }
};
```

Reinvent the wheel!!

```
template<class T>
class Range {     // simple value type
  T low, value, high;     // Invariant: low <= value < high
  void check(T v) { if !(low <= v && v < high) throw RangeErr(); }
public:
  Range(T lo, T val, T hi) : low(lo), value(val), high(hi) {}
  Range& operator=(const Range& r) { check(r.value); value = r.value;
         return *this; }
  Range& operator=(const T& v) { check(v); value = v; return *this;}
  operator T() const { return value; } // call copy ctor
};
```

using Range concept- Template version

- The name of a class template followed by a type bracketed by < > is the name of a class (as defined by the template) and can be used exactly like other class names.

```
void f()
{
    Range r; // error: Range is not a type
    Range<int> ri(0, 1, 2); // ok: Range<int> is a type
    Range<double> rd(0, 2.7182, 3.1416); // ok: another type
    Range<Point> rp(Point(), Point(1, 2), Point(-1,2)); // ok: yet another type
    Range<std::complex<double>> rc(std::complex(), std::complex(0, 1), std::complex(1, 0));
    // error: no relop for complex numbers
}
```

- For using a type as a template argument for range, it has to support <= and < operators and it must have a copy constructor and assignment operator. In other words, it should be *Less than comparable, Less than or equal comparable*, *Copy constructible,* and *Assignable*.
- There is no requirement that different arguments for the same template parameter should be related by inheritance.
- Templates are a compile-time mechanism so that their use incurs no runtime overhead compared to "handwritten code."

template Instantiation

template Instantiation

- The process of generating a class or a function from a template plus a template argument list is often called *template instantiation*.

template Instantiation

- The process of generating a class or a function from a template plus a template argument list is often called *template instantiation*.

template Instantiation

- The process of generating a class or a function from a template plus a template argument list is often called *template instantiation*.

- The process of replacing template parameters by concrete types is called *instantiation*.

template Instantiation

- The process of generating a class or a function from a template plus a template argument list is often called *template instantiation*.
- The process of replacing template parameters by concrete types is called *instantiation*.
- Instantiation is a process, rather than type name substitution.

template Instantiation

- The process of generating a class or a function from a template plus a template argument list is often called *template instantiation*.

- The process of replacing template parameters by concrete types is called *instantiation*.

- Instantiation is a process, rather than type name substitution.

- Template instantiation: more details
  - From a class template and a set of template arguments, the compiler needs to generate the definition of a class and the definitions of those of its member functions that were used in the program (and only those). From a template and a set of template arguments, a function needs to be generated.

template Instantiation

- The process of generating a class or a function from a template plus a template argument list is often called *template instantiation*.

- The process of replacing template parameters by concrete types is called *instantiation*.

- Instantiation is a process, rather than type name substitution.

- Template instantiation: more details
  - From a class template and a set of template arguments, the compiler needs to generate the definition of a class and the definitions of those of its member functions that were used in the program (and only those). From a template and a set of template arguments, a function needs to be generated.

- Template instantiation vs. Class instantiation.

template Instantiation

- The process of generating a class or a function from a template plus a template argument list is often called *template instantiation*.

- The process of replacing template parameters by concrete types is called *instantiation*.

- Instantiation is a process, rather than type name substitution.

- Template instantiation: more details
  - From a class template and a set of template arguments, the compiler needs to generate the definition of a class and the definitions of those of its member functions that were used in the program (and only those). From a template and a set of template arguments, a function needs to be generated.

- Template instantiation vs. Class instantiation.

Generic programming

template Instantiation

- The process of generating a class or a function from a template plus a template argument list is often called *template instantiation*.

- The process of replacing template parameters by concrete types is called *instantiation*.

- Instantiation is a process, rather than type name substitution.

- Template instantiation: more details
  - From a class template and a set of template arguments, the compiler needs to generate the definition of a class and the definitions of those of its member functions that were used in the program (and only those). From a template and a set of template arguments, a function needs to be generated.

- Template instantiation vs. Class instantiation.

Generic programming

Object-oriented programming

template Instantiation: an example

template Instantiation: an example

1

```
template<class T>
inline T const& min(const T& a, const T& b)
{
  return a < b ? a : b;
}
```

template Instantiation: an example

```
template<class T>
inline T const& min(const T& a, const T& b)
{
  return a < b ? a : b;
}
```

1

```
min(1, 2);
```

2

template Instantiation: an example

1

```
template<class T>
inline T const& min(const T& a, const T& b)
{
  return a < b ? a : b;
}
```

2

```
min(1, 2);
```

3

```
inline int const& min(const int& a, const int& b)
{
  return a < b ? a : b;
}
```

template Instantiation: an example

1

```
template<class T>
inline T const& min(const T& a, const T& b)
{
  return a < b ? a : b;
}
```

2

```
min(1, 2);
```

3

```
inline int const& min(const int& a, const int& b)
{
  return a < b ? a : b;
}
```

4

```
template<class Type>
class Holder { // a simple object holder
  Type a;
  // …
};
```

template Instantiation: an example

1

```
template<class T>
inline T const& min(const T& a, const T& b)
{
  return a < b ? a : b;
}
```

2

```
min(1, 2);
```

3

```
inline int const& min(const int& a, const int& b)
{
  return a < b ? a : b;
}
```

4

```
template<class Type>
class Holder { // a simple object holder
  Type a;
  // …
};
```

5

```
Holder<Point> hp;
```

template Instantiation: an example

1

```
template<class T>
inline T const& min(const T& a, const T& b)
{
  return a < b ? a : b;
}
```

2

```
min(1, 2);
```

3

```
inline int const& min(const int& a, const int& b)
{
  return a < b ? a : b;
}
```

4

```
template<class Type>
class Holder { // a simple object holder
  Type a;
  // …
};
```

5

```
Holder<Point> hp;
```

6

```
class Holder { // a simple object holder
  Point a;
  // …
};
```

template Instantiation

template Instantiation

- Compiler generates type-specific classes/functions.

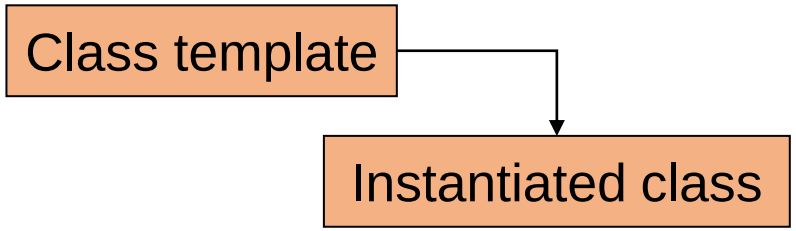
template Instantiation

- Compiler generates type-specific classes/functions.

Class template

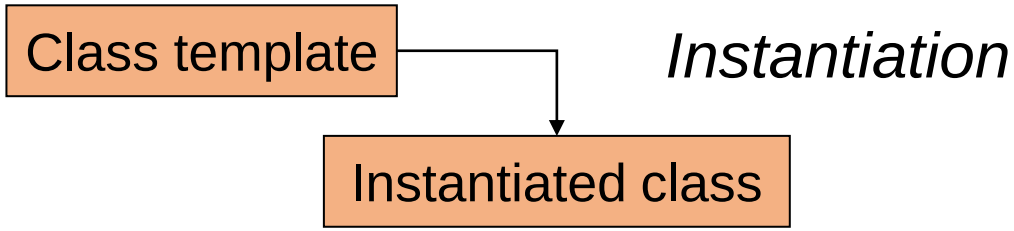
template Instantiation

- Compiler generates type-specific classes/functions.



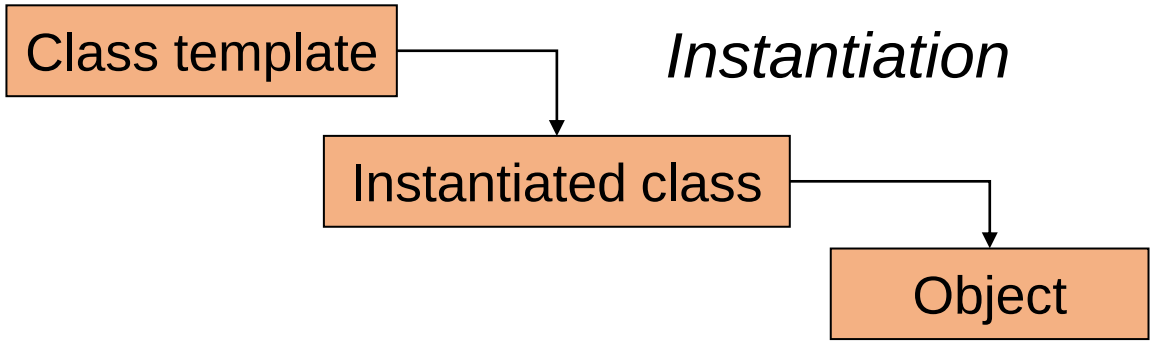
template Instantiation

- Compiler generates type-specific classes/functions.



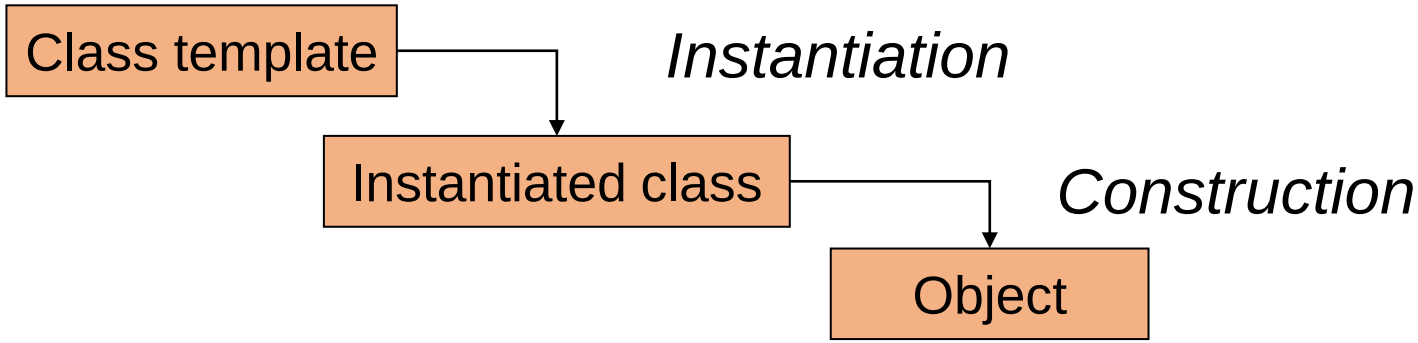
template Instantiation

- Compiler generates type-specific classes/functions.



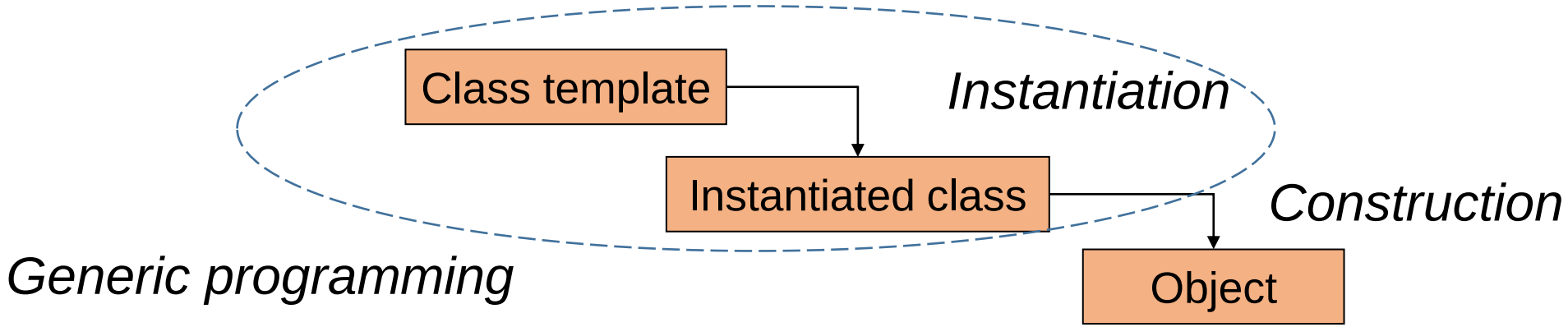
template Instantiation

- Compiler generates type-specific classes/functions.



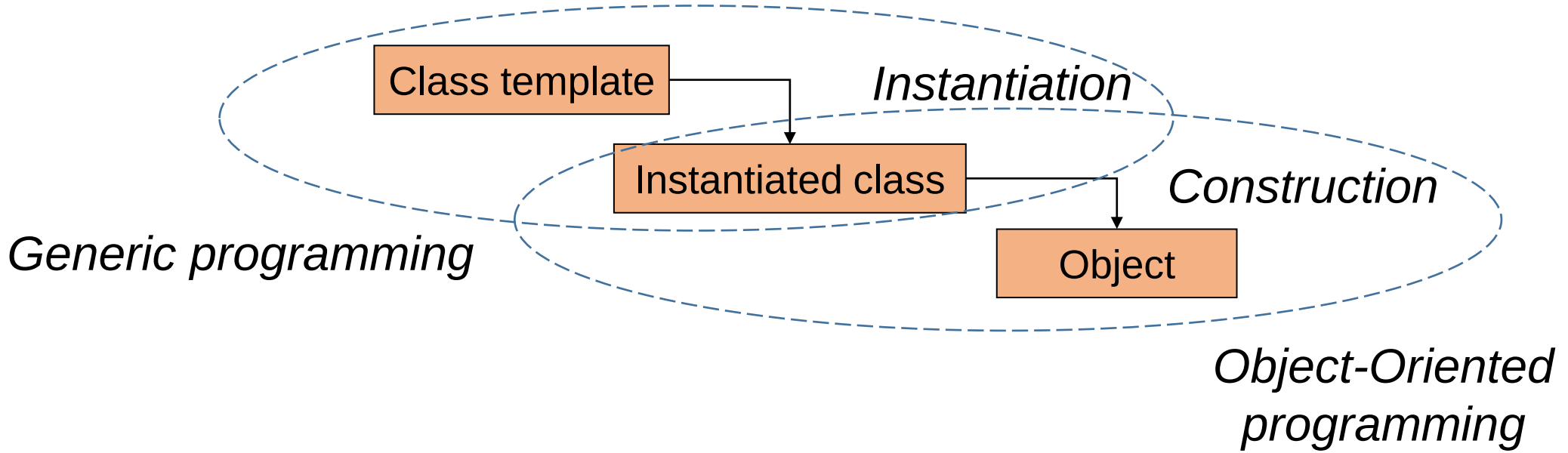
template Instantiation

- Compiler generates type-specific classes/functions.



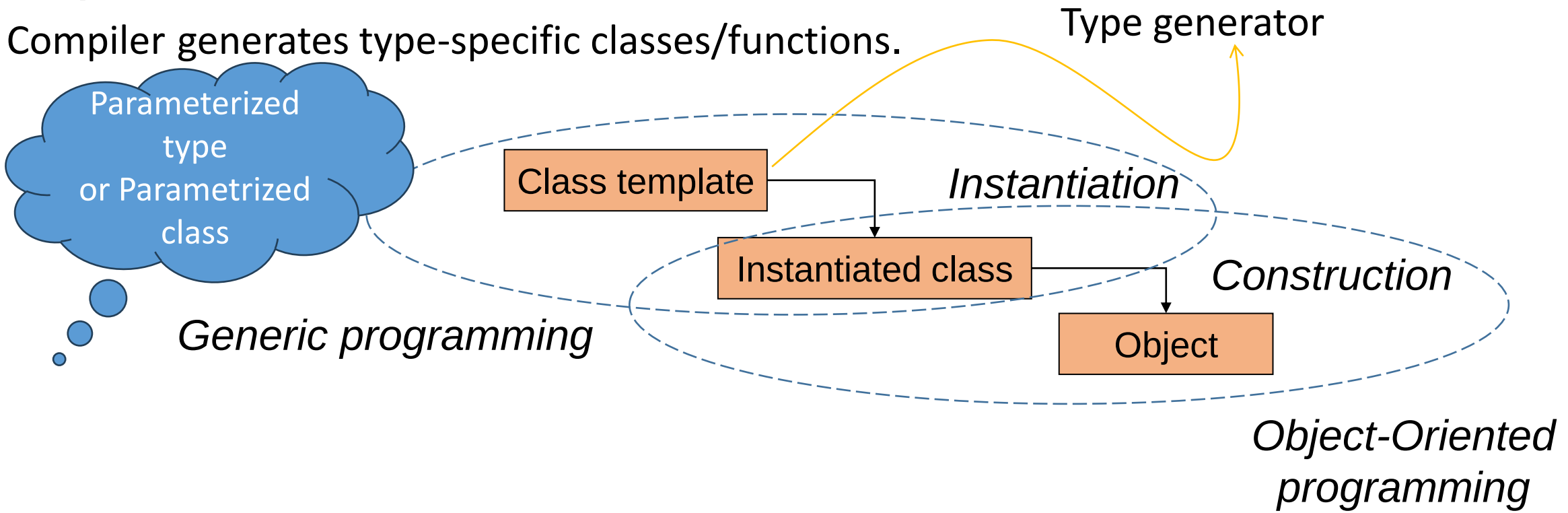
template Instantiation

- Compiler generates type-specific classes/functions.



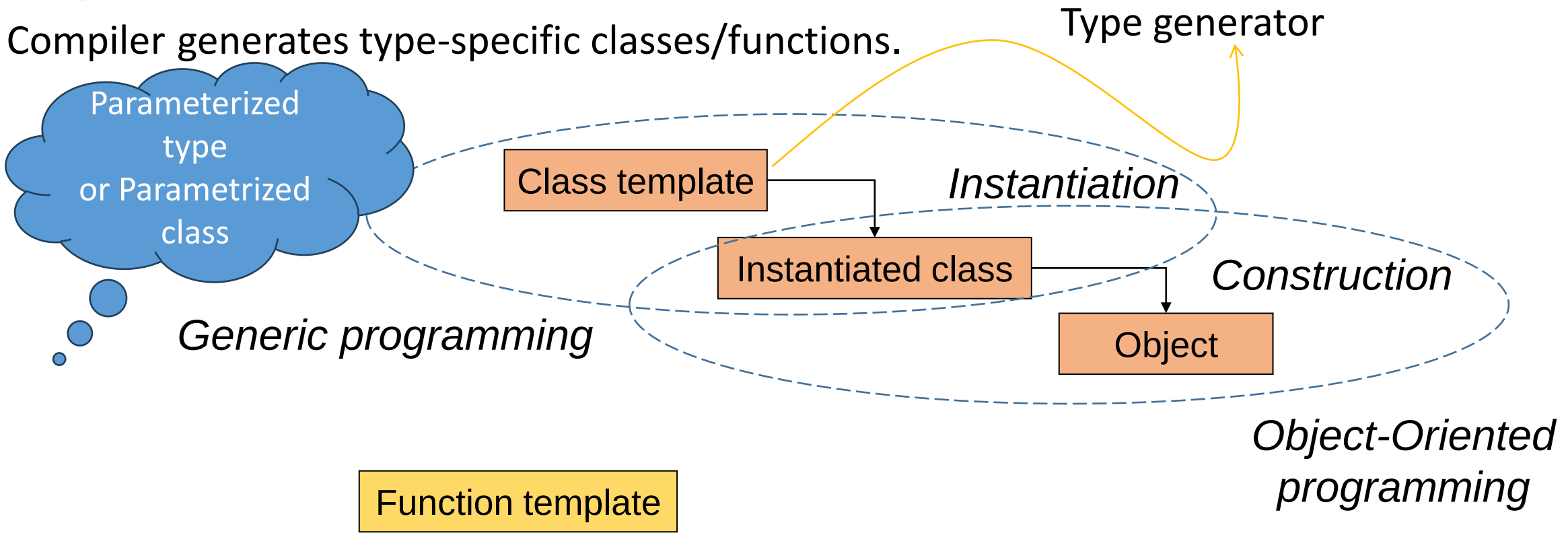
template Instantiation

- Compiler generates type-specific classes/functions.



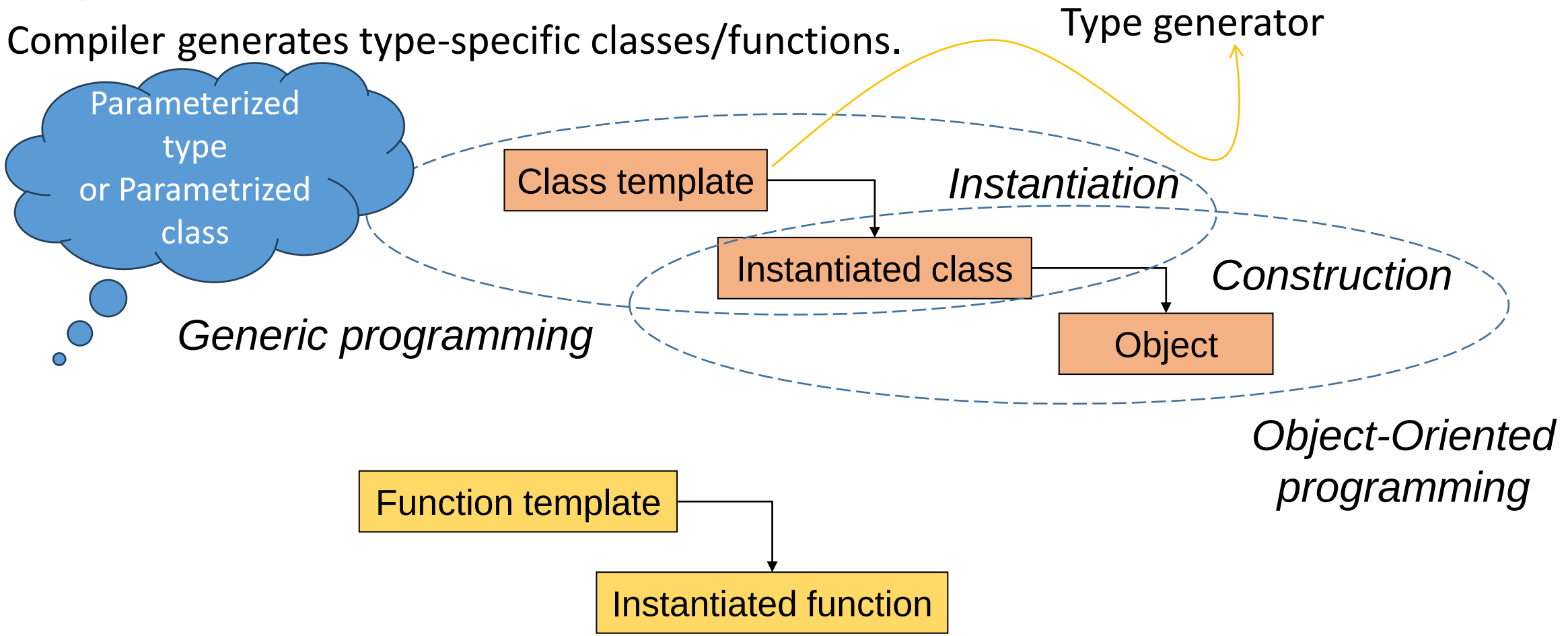
template Instantiation

- Compiler generates type-specific classes/functions.



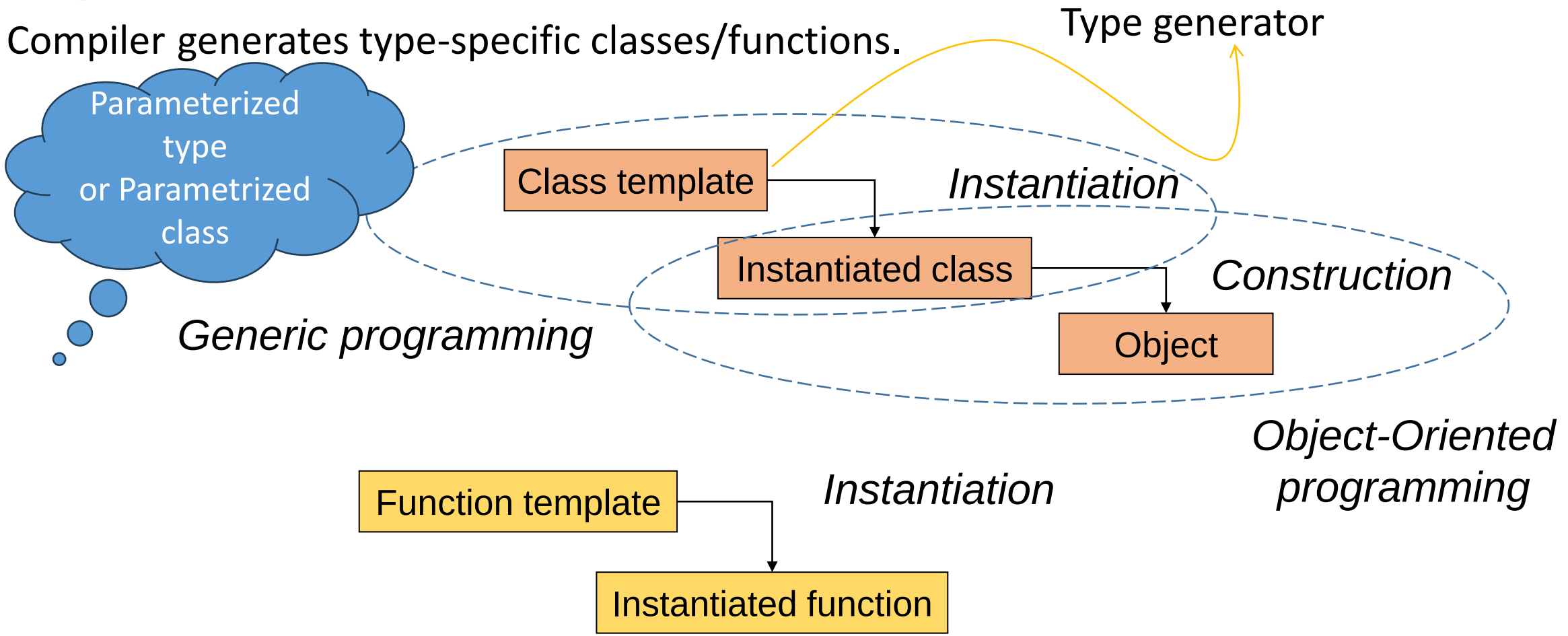
template Instantiation

- Compiler generates type-specific classes/functions.



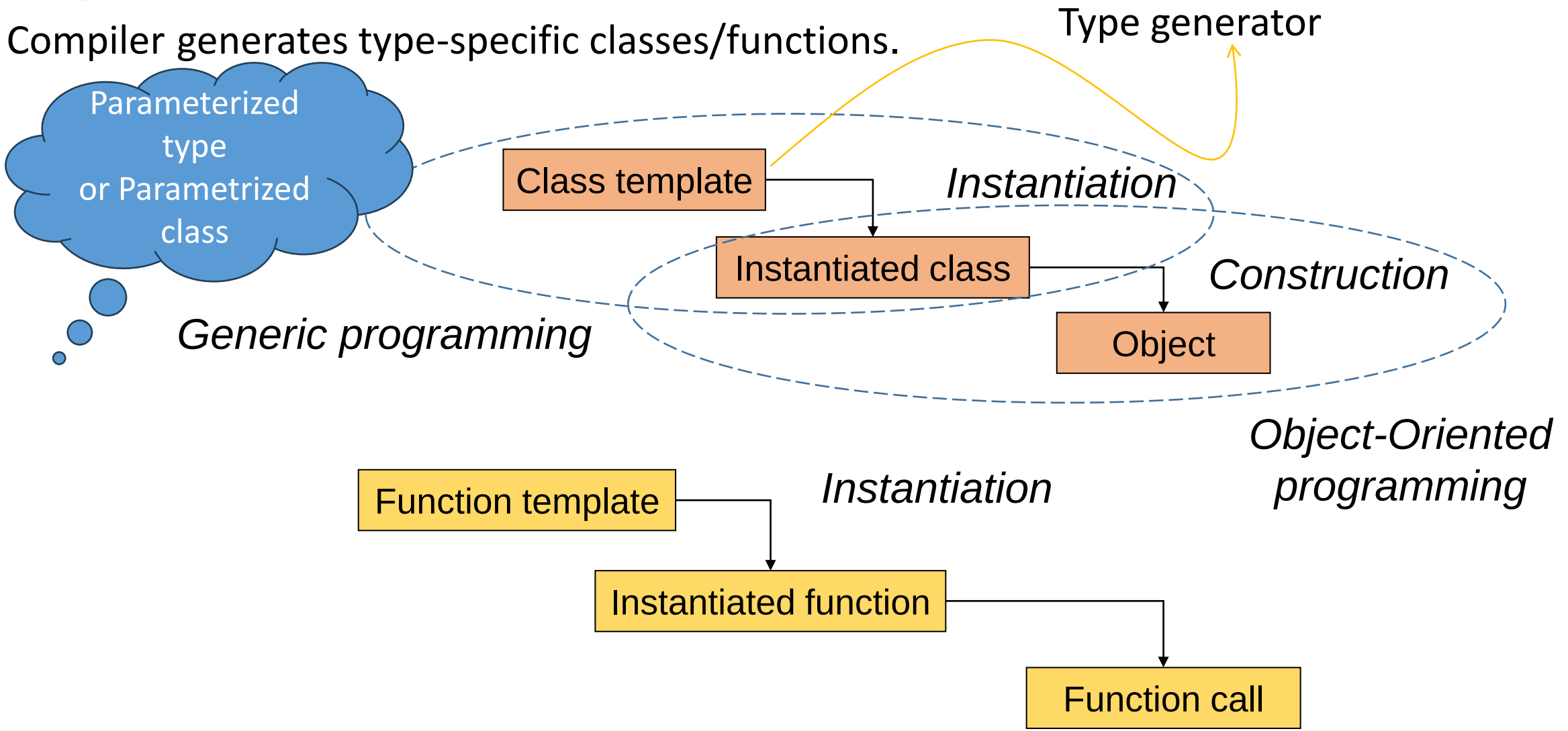
template Instantiation

- Compiler generates type-specific classes/functions.



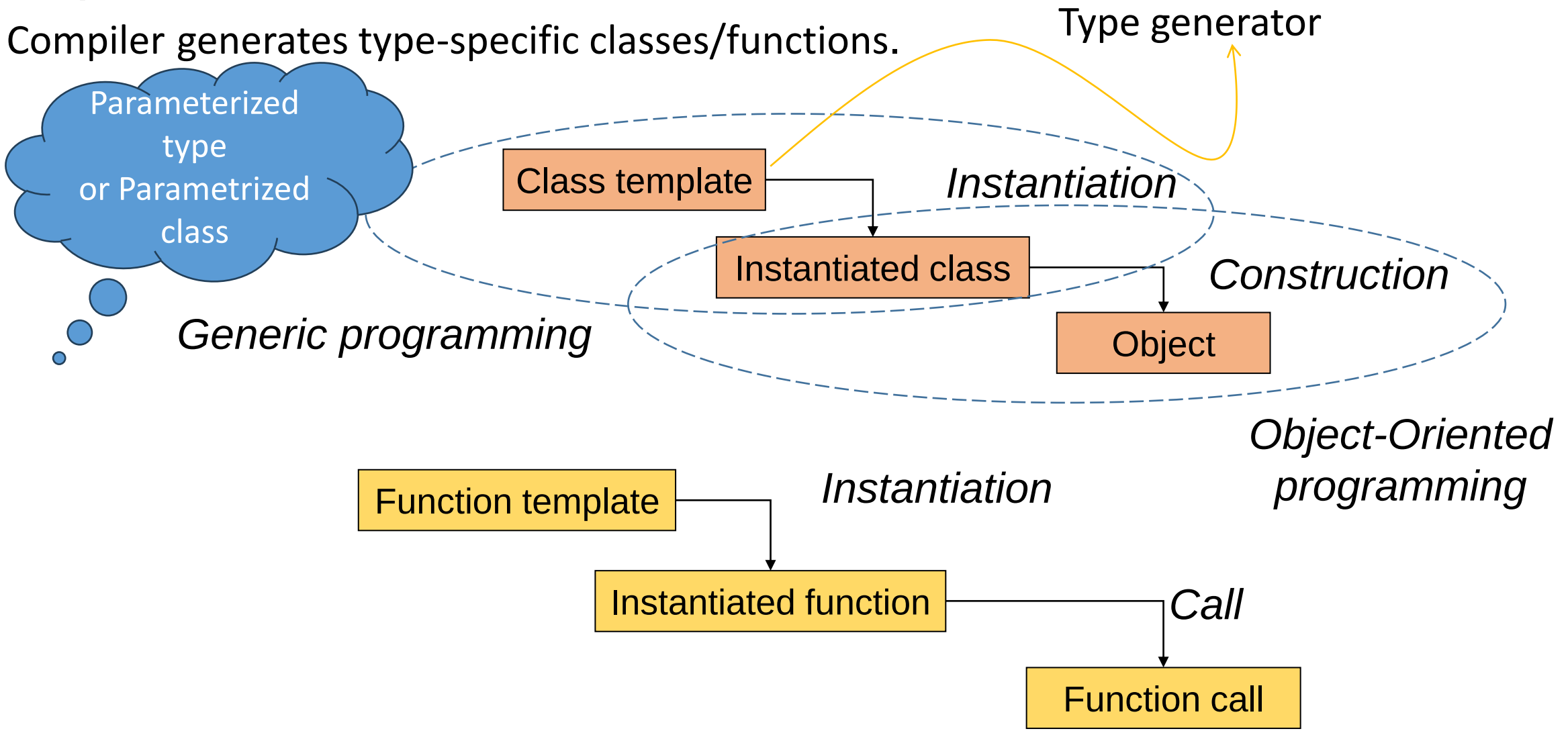
template Instantiation

- Compiler generates type-specific classes/functions.



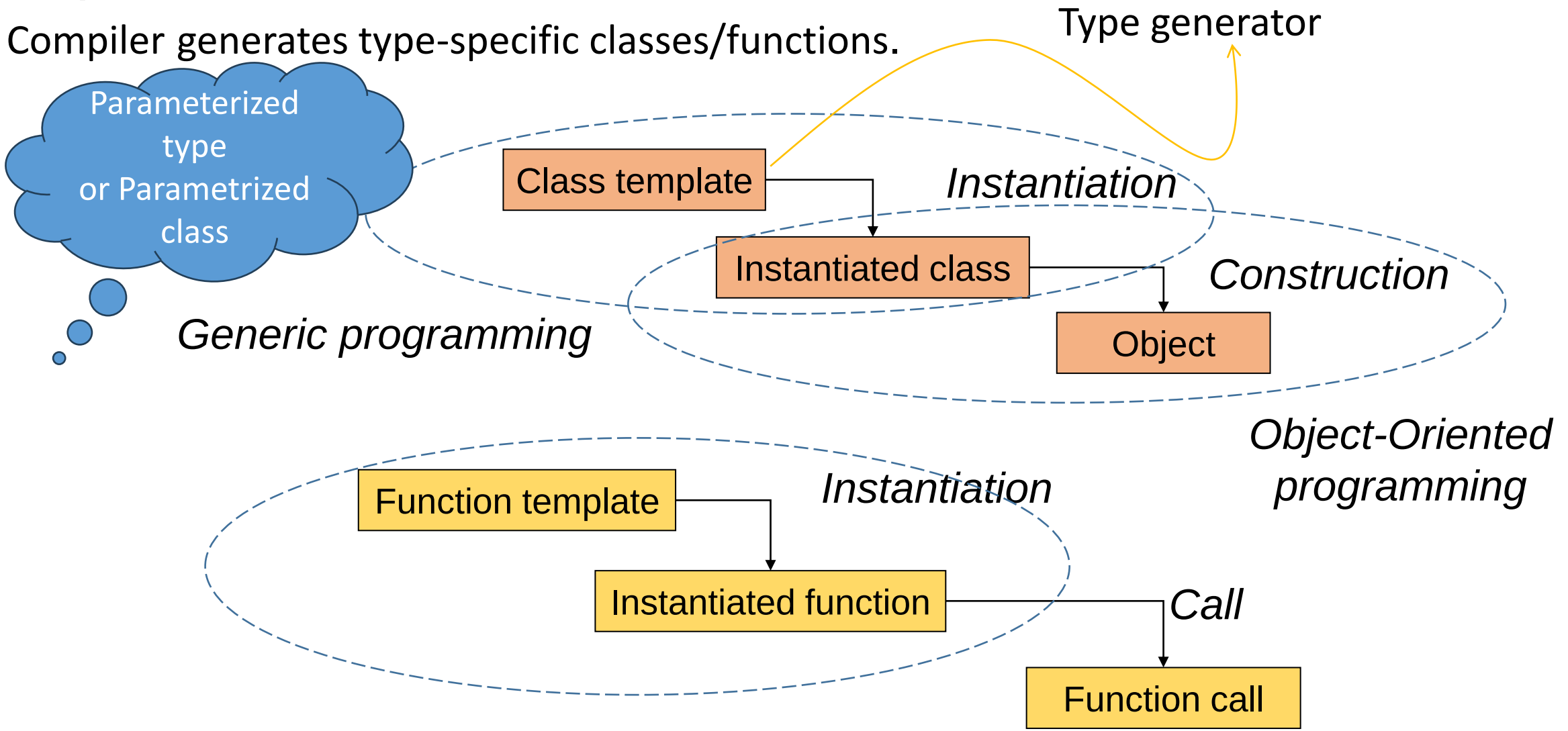
template Instantiation

- Compiler generates type-specific classes/functions.



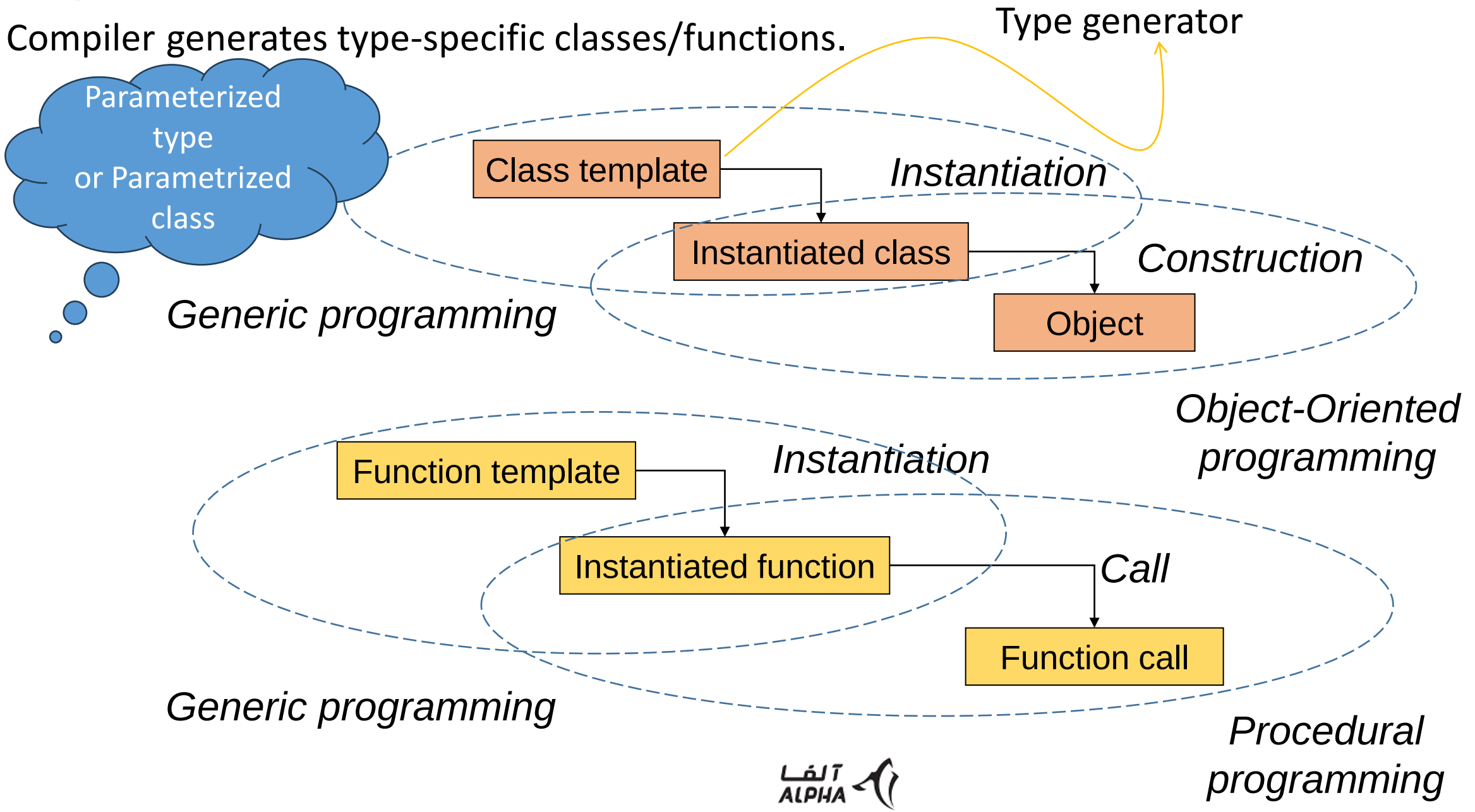
template Instantiation

- Compiler generates type-specific classes/functions.



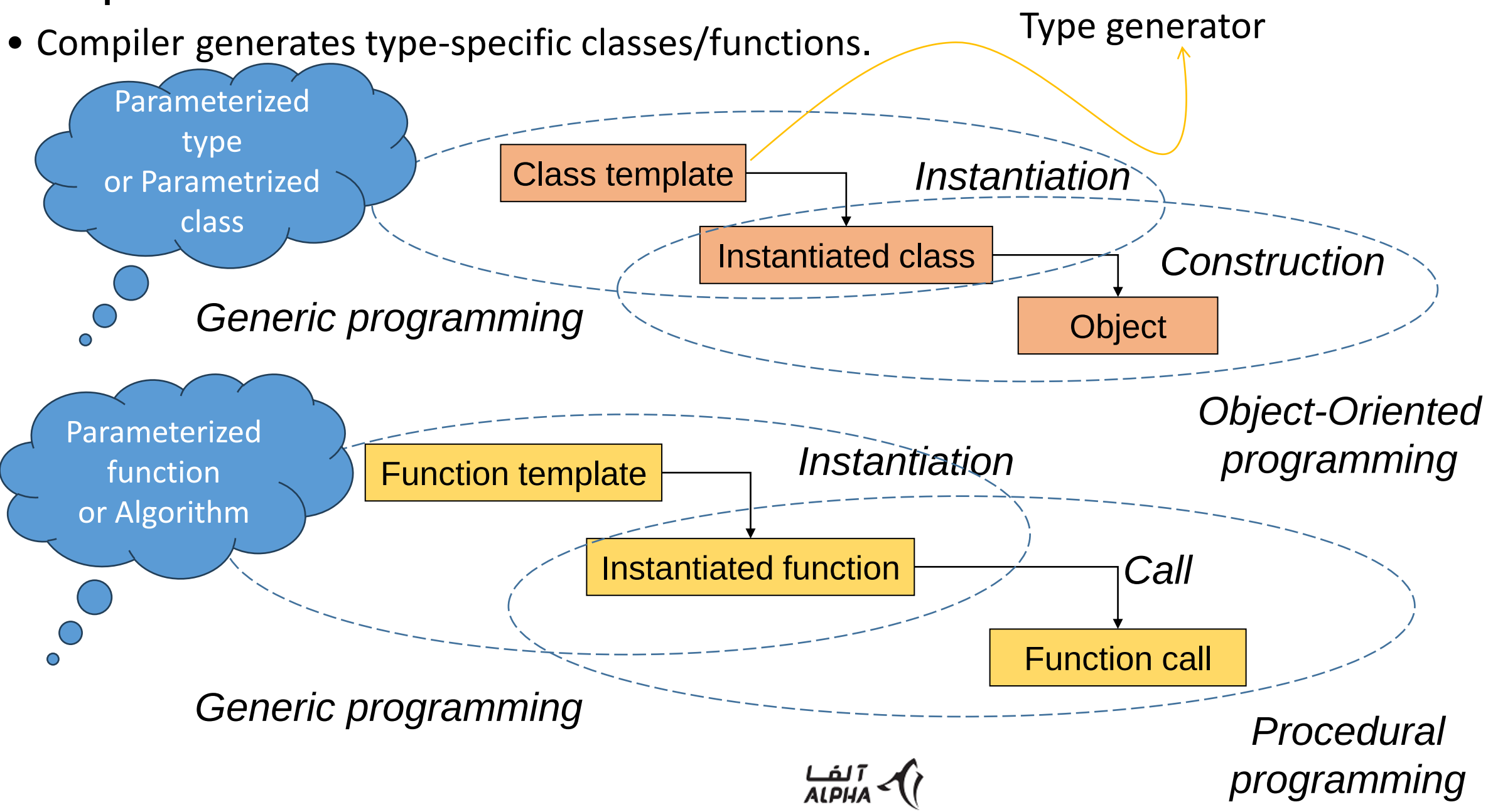
template Instantiation

- Compiler generates type-specific classes/functions.



template Instantiation

- Compiler generates type-specific classes/functions.



Templates- old terminologies

function template *vs.* template function

class template *vs.* template class

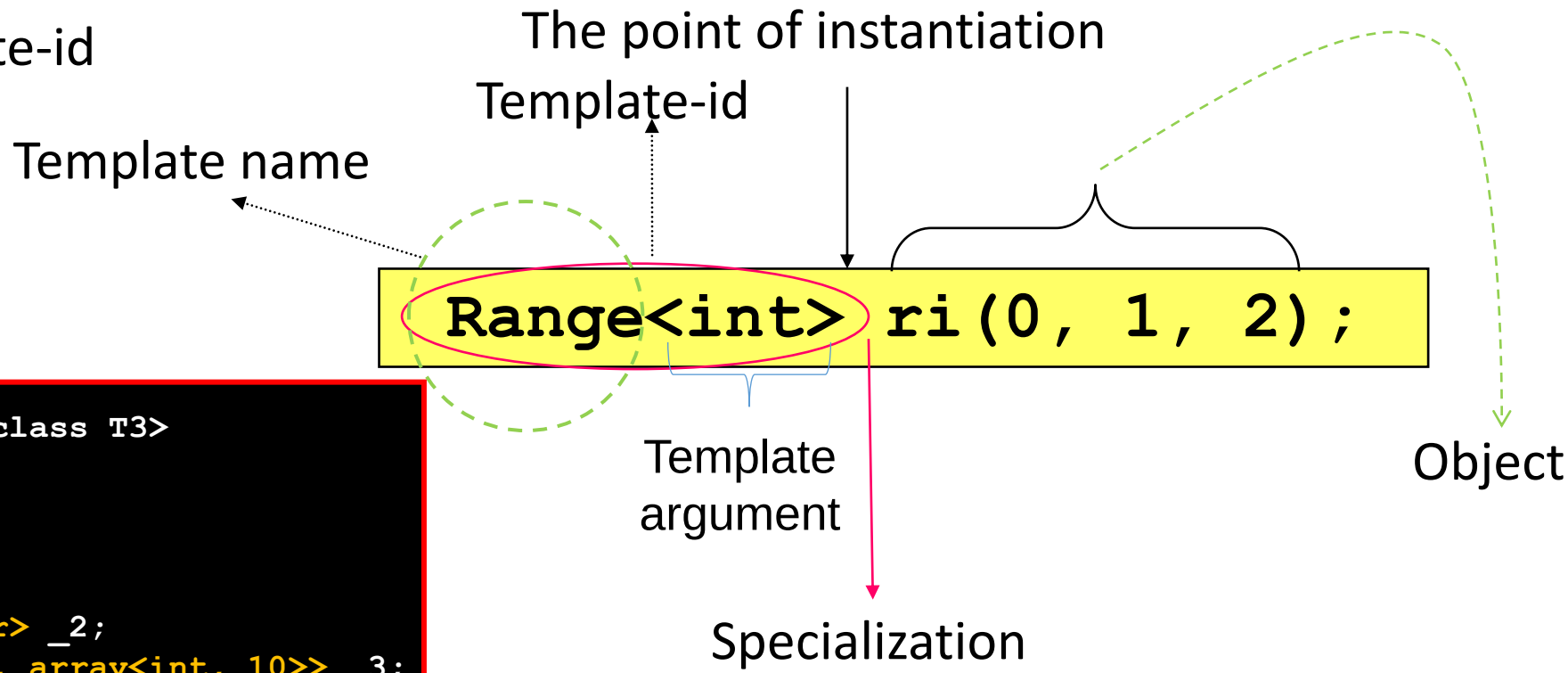
- Nathan Myers:

  - Function template: A template that describes how to create a set of functions.
  - Template function: A function generated (or specialized) from such a template.
  - Class template: A template that describes how to create a set of classes.
  - Template class: A class generated (or specialized) from such a template.

- This is an old and deprecated terminology.
- Better terminology: Instantiated function and Instantiated class

Specialization

- The process of generating a class or a function from a template plus a template argument list is often called *template instantiation*.
- A version of a template for a specific template argument list is called a *specialization*.
- Instantiated template = Specialization
- Specialization A.K.A template-id



- Template-id

```
template<class T1, class T2, class T3>
struct Triple {
  T1 t1; T2 t2; T3 t3;
};
Triple<int, double, int> _1;
Triple<std::string, bool, char> _2;
Triple<list<int>, vector<int>, array<int, 10>> _3;
Triple<bool, bool, Triple<bool, bool, bool>> _4;
```

Template-id

Thanks for your patience …

A man who asks a question is a fool for minute,
The man who does not ask, is a fool for a life.
- Confucius

Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer.
- Bjarne Stroustrup *programming Principles and Practice Using C++, page 4.*

There is no stupid question, but there is stupid answer.
- Howard Hinnant