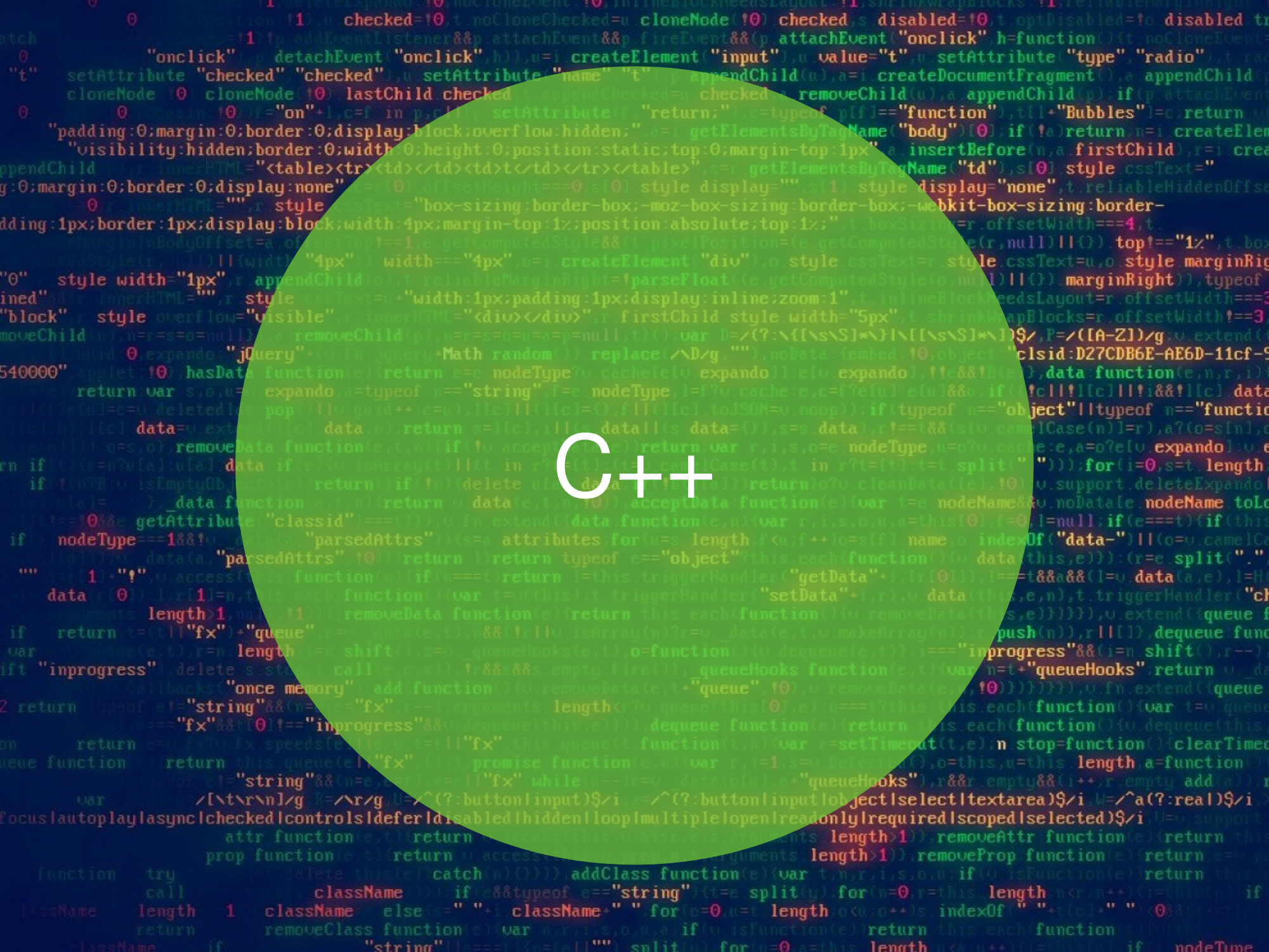
Introduction to C++

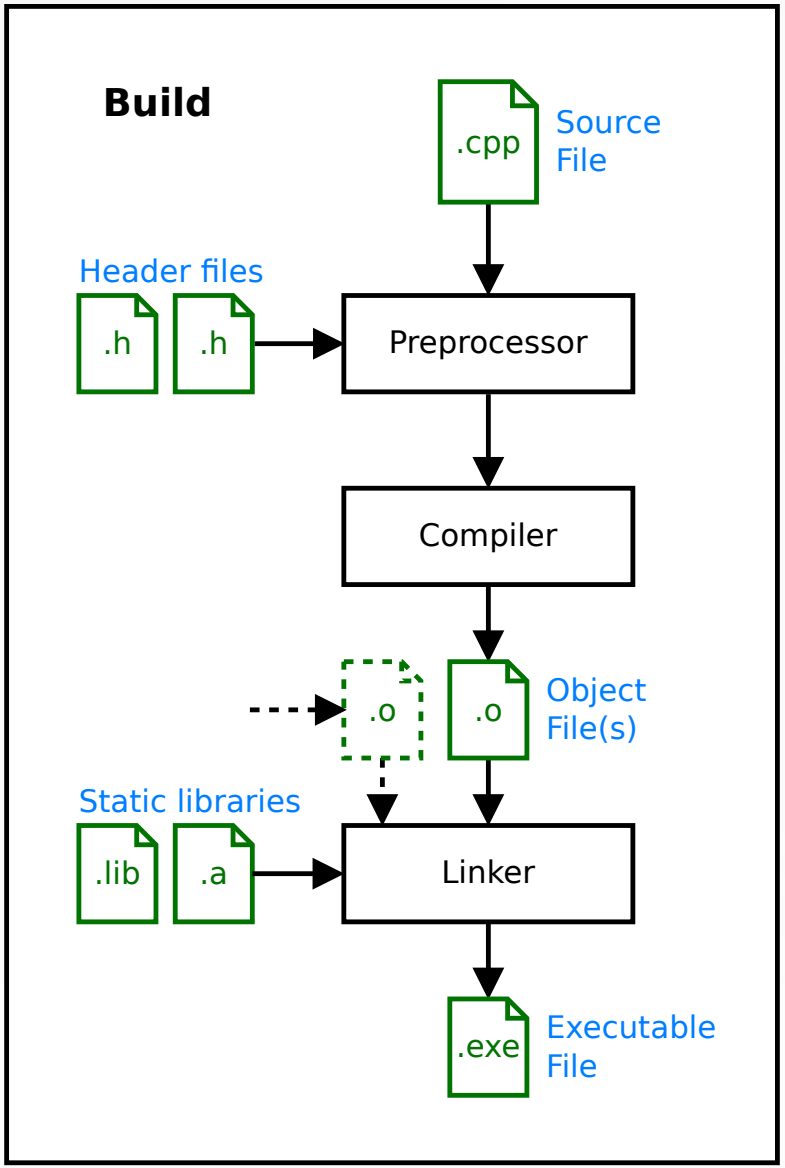
Alexandre Kaspar • EPFL / MIT • akaspar@mit.edu

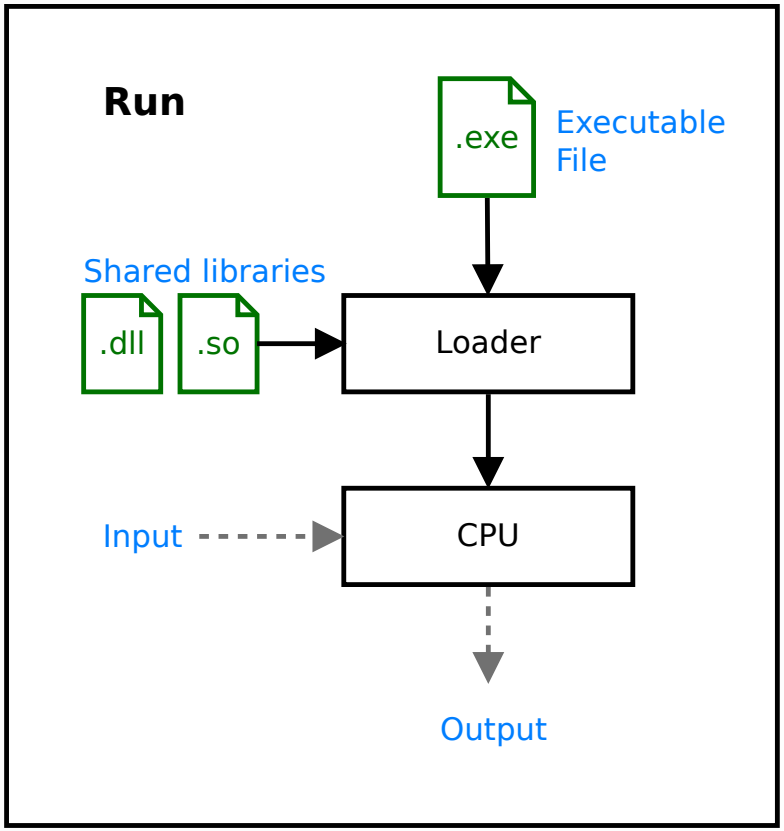

C++

History

- General-purpose programming language
- Created in 1979 by Bjarne Stroustrup
- Extension of C that includes
- Object-oriented programming, templating, polymorphism, operator overloading . . .

Build & run


Build
.cpp
Source File
Header files
.h
.h
Preprocessor
Compiler
.o
.o
Object File(s)
Static libraries
.lib
.a
Linker
.exe
Executable File


Run
.exe
Executable File
Shared libraries
.dll
.so
Loader
Input
CPU
Output

Compile, link and load

- The *compiler* translates source files (.cpp) into object files (.o).
- The *linker* gathers information to link object files (.o) together with static libraries, producing an executable file (.exe).
- The *loader* connects the executable file (.exe) with shared libraries (.so).


HELLO WORLD!

Hello World

```
1  #include<iostream>
2  // A comment
3  int main(int argc, char **argv){
4      std::cout << "Hello world!\n";
5      return 0;
6  }
```

Outputs ”Hello world!” to the console and exits.

Pre-processor instructions

```
1  #include<iostream>
```

Lines starting with a '#' (pound) are instructions parsed by the *pre-processor* such as

- **#include** specifies a file to be included at this line.
- **#if, #ifdef, #endif** conditionally uses a block of code.
- **#define** creates an alias for an expression.

Comments

```
2  // A comment (up to the end of the line)
```

or

```
2  /* A comment
3  possibly spanning
4  multiple lines */
```

are two types of comments, which the compiler ignores.

Main function

```
3  int main(int argc, char **argv){
4      ...
5      return 0;
6  }
```

defines a function named `main` which

- takes two arguments `argc` and `argv`,
- returns an integer value (`int` in front).

Main function (2)

```
3  int main(int argc, char **argv){
4      ...
5      return 0;
6  }
```

The `main` function is called when the program starts, with

- `argc`, the number of arguments
- `argv`, the arguments (array of strings)

and returns an integer code (0 = success).

Output statement

```
4  std::cout << "Hello␣world!\n";
```

- `std` is the standard *namespace*.
- `cout` is a *Stream* object.
- `<<` is the output operator of *Stream*.
- `"Hello world"` is the string being output.

BASICS

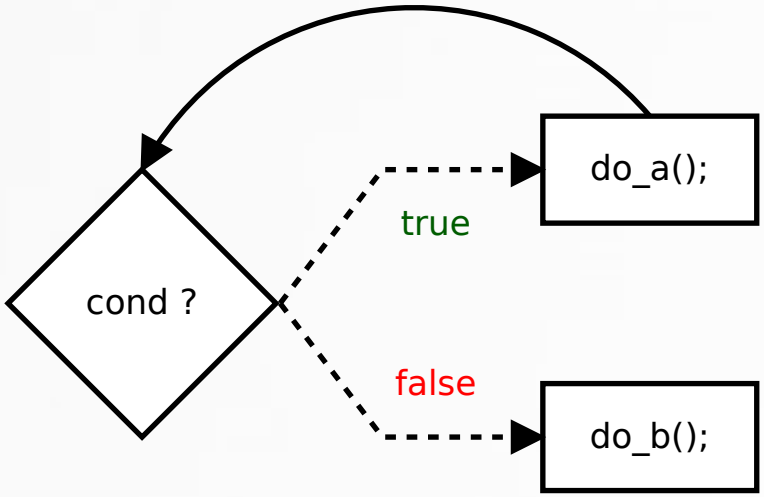
Declaration and definition

```
void process(int x); // declare a function
int x; // declare a variable
x = 10; // assign a value
process(x);
// declaration and definition
int sum(int a, int b){ return a + b; }
// declaration and assignment
int y = sum(x, 2);
```

Every variable, type or function must be declared before being used. They can then be defined anywhere.
Note: variables automatically receive a default value corresponding to 0.

Flow control

```
while(cond){
    do_a();
}
do_b();
```



The usual conditional blocks are available

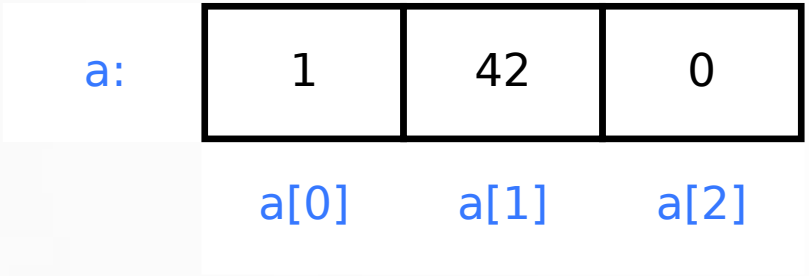
- `if`, `else if`, `else` and `switch`

as well as loop structures

- `for`, `while` and `do while`

Arrays

```
1  int a[3] = { 1, 0 };
2  a[1] = 42;
3  // a[2] == 0
```



Arrays are continuous blocks of memory that store multiple elements of a same type. They use 0-based indexing.

POINTERS

Passing by value

```
void swap_wrong(int x, int y){
    int tmp = x;
    x = y;
    y = tmp;
}
int a = 1, b = 2;
swap_wrong(a, b);
// did not work! a=1, b=2
```

Function arguments are **passed by value** (copy of value). We need pointers to modify `a` and `b` in the code above.

Using pointers

```
void swap_ptr(int *x, int *y){
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
int a = 1, b = 2;
swap_ptr(&a, &b);
// now a=2, b=1
```

`&a` uses the **reference** operator to get the adresse of `a`.
`*x` uses the **dereference** operator to get / set the value at the adress given by `x`.

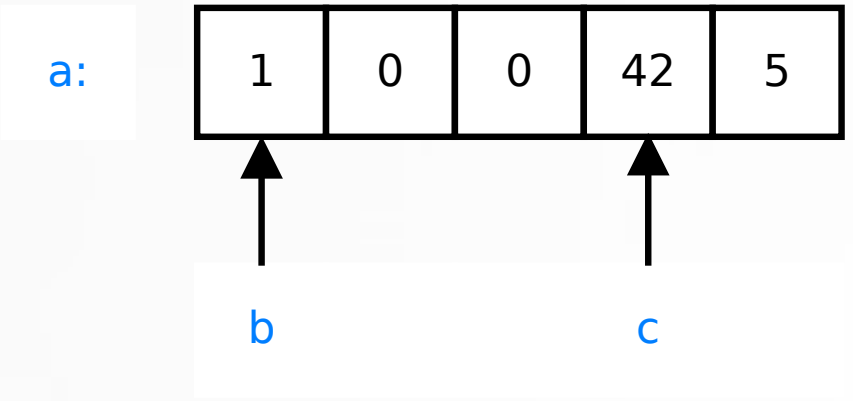
References

```
// passed by reference
void swap_ref(int &x, int &y){
    int tmp = x;
    x = y;
    y = tmp;
}
int a = 1, b = 2;
swap_ref(a, b);
// good! a=2, b=1
```

References are syntactic sugar for pointers when passing variables to functions or retrieving the value they return.

Pointer arithmetic

```
1  int a[5] = { 1 };
2  int *b = &a[0];
3  int *c = b + 3;
4  *c = 42; // or c[0]
5  c[1] = 5;
```



- Accessing a pointer as an array
- Pointer arithmetic according to array indexing

Warning! A pointer value is the byte address and types have different sizes which can be found using `sizeof(mytype)`.


TEMPLATING

Templating

```
template<typename T>
T sum(T a, T b){
    T c = a + b;
    return c;
}
```

Function (and class) *templates* provides a way to abstract the notion of type.
Here, the type `T` is resolved by the compiler when `sum` is used.
It only requires an operator `+` to work*.

Templating (2)

```
int c = sum(1, 2);
float d = sum(1.0f, 2.0f);
float e = sum<float>(1, 2.0f);
```

The type `T` can sometimes by inferred. When it cannot, one must specify it.

Templating (3)

```
template<>
bool sum<bool>(bool a, bool b){
    return a || b; // a or b
}
bool a = true, b = false;
bool c = sum(a, b);
// c == true
```

Templates can be specialized for a given type.


CLASSES

Classes

```
// declaration
template <typename T, int dim>
struct vec {
    typedef vec<T, dim> this_type;
    T val[dim];
    vec(T a, T b);
    T sqLength() const;
    static this_type constant(T a);
};
```

Classes (keyword `class`) and structures (keyword `struct`) describe custom types made of properties (`val`) and methods (`sqLength`).

Classes (2)

```
// definitions
template <typename T, int dim>
T vec<T, dim>::sqLength() {
    T sum = 0;
    for(int i = 0; i < dim; ++i)
        sum += val[i] * val[i];
    return sum;
}
```

Definition of methods (including the constructor `vec` and/or the possible destructor `~vec` can be done separately.

Classes (3)

```
// type alias
typedef vec<float, 2> vec2f;

// class usage
vec2f a = vec2f::constant(2); // (2, 2)
float d = a.sqLength(); // d=8
```

Static properties and methods are called with the namespace operator (`::`) whereas instance members are accessed using the dot operator.

Operator overloading

```
vec2f operator +(const vec2f &v1,
                 const vec2f &v2){
    vec2f x;
    for(int i = 0; i < 2; ++i)
        x.val[i] = v1.val[i] + v2.val[i];
    return x;
}

// usage
vec2f a, b;
// hidden: initialization of a and b
vec2f c = a + b;
```

STACK / HEAP

References

```
// passed by reference
void swap_ref(int &x, int &y){
    int tmp = x;
    x = y;
    y = tmp;
}
int a = 1, b = 2;
swap_ref(a, b);
// good! a=2, b=1
```

References are syntactic sugar for pointers when passing variables to functions or retrieving the value they return.


POLYMORPHISM

References

```
// passed by reference
void swap_ref(int &x, int &y){
    int tmp = x;
    x = y;
    y = tmp;
}
int a = 1, b = 2;
swap_ref(a, b);
// good! a=2, b=1
```

References are syntactic sugar for pointers when passing variables to functions or retrieving the value they return.


CONVERSIONS

References

```
// passed by reference
void swap_ref(int &x, int &y){
    int tmp = x;
    x = y;
    y = tmp;
}
int a = 1, b = 2;
swap_ref(a, b);
// good! a=2, b=1
```

References are syntactic sugar for pointers when passing variables to functions or retrieving the value they return.

ADVANCED C++

Templating

```
template< typename T >
T sum(T a, T b){
    T c = a + b;
    return c;
}
```

Function (and class) *templates* provides a way to abstract the notion of type.
Here, the type `T` is resolved by the compiler when `sum` is used.
It only requires an operator `+` to work*.

Templating (2)

```
int c = sum(1, 2);
float d = sum(1.0f, 2.0f);
float e = sum<float>(1, 2.0f);
```

The type `T` can sometimes by inferred. When it cannot, one must specify it.

Classes

```
// declaration
template <typename T, int dim>
struct vec {
    typedef vec<T, dim> this_type;
    T val[dim];
    vec(T a, T b);
    T sqLength() const;
    static this_type constant(T a);
};
```

Classes (keyword `class`) and structures (keyword `struct`) describe custom types made of properties (`val`) and methods (`sqLength`).

Classes (2)

```
// definitions
template <typename T, int dim>
T vec<T, dim>::sqLength() {
    T sum = 0;
    for(int i = 0; i < dim; ++i)
        sum += val[i] * val[i];
    return sum;
}
```

Definition of methods (including the constructor `vec` and/or its possible destructor `~vec` can be done separately.

Classes (3)

```
// type alias
typedef vec<float, 2> vec2f;

// class usage
vec2f a = vec2f::constant(2); // (2, 2)
float d = a.sqLength(); // d=8
```

Static properties and methods are called with the namespace operator (`::`) whereas instance properties and methods are accessed with the dot operator.

Operator overloading

```
vec2f operator +(const vec2f &v1,
                 const vec2f &v2){
    vec2f x;
    for(int i = 0; i < 2; ++i)
        x.val[i] = v1.val[i] + v2.val[i];
    return x;
}

// usage
vec2f a, b;
// hidden: initialization of a and b
vec2f c = a + b;
```

THANK YOU

SOURCES

- Introduction to C++, MIT OpenCourseWare
- A Quick Introduction to C++, Thomas Anderson
- C/C++ Tutorials, Chua Hock-Chuan
- C++11 Faq, Bjarne Stroustrup