

Classes



Classes

In C++ classes are the main kind of user-defined type.

Informal specification of a class definition:

```
class-keyword name {  
    member-specification  
};
```

- *class-keyword* is either **struct** or **class**
- *name* can be any valid identifier (like for variables, functions, etc.)
- *member-specification* is a list of declarations, mainly variables (“data members”), functions (“member functions”), and types (“nested types”)
- The trailing semicolon is mandatory!



Data Members

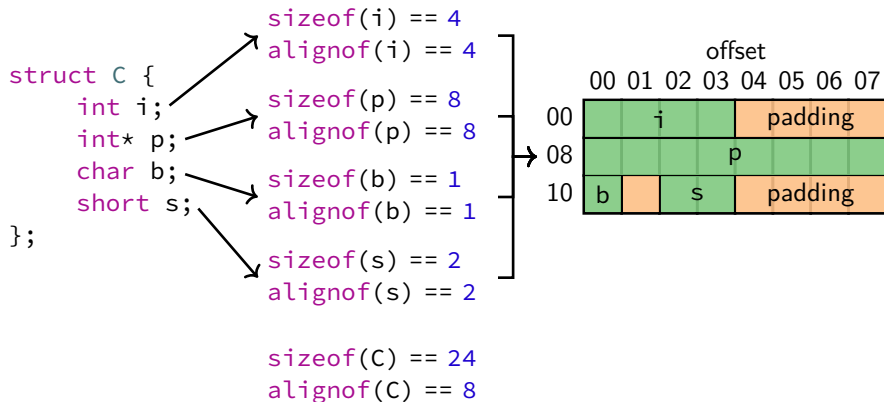
- Declarations of data members are variable declarations
- `extern` is not allowed
- Declarations without `static` are called *non-static* data members, otherwise they are *static* data members
- `thread_local` is only allowed for static data members
- Declaration must have a *complete type* (see later slide)
- Name of the declaration must differ from the class name and must be unique within the class
- Non-static data members can have a *default value*

```
struct Foo {  
    // non-static data members:  
    int a = 123;  
    float& b;  
    const char c;  
    // static data members:  
    static int s;  
    thread_local static int t;  
};
```

Memory Layout of Data Members

- Every type has a size and an alignment requirement
- To be compatible between different compilers and programming languages (mainly C), the memory layout of objects of class type is fixed
- Non-static data members appear in memory by the order of their declarations
- Size and alignment of each data-member is accounted for → leads to “gaps” in the object, called *padding bytes*
- Alignment of a class type is equal to the largest alignment of all non-static data members
- Size of a class type is at least the sum of all sizes of all non-static data members and at least 1
- static data members are stored separately

Size, Alignment and Padding



Reordering the member variables in the order p, i, s, b would lead to `sizeof(C) == 16!`

In general: Order member variables by decreasing alignment to get the fewest padding bytes.



Member Functions

- Declarations of member functions are like regular function declarations
- Just like for data members, there are non-static and static (with the `static` specifier) member functions
- Non-static member functions can be *const-qualified* (with `const`) or *ref-qualified* (with `const&`, `&`, or `&&`)
- Non-static member functions can be `virtual`
- There are some member functions with special functions:
 - Constructor and destructor
 - Overloaded operators

```
struct Foo {  
    void foo(); // non-static member function  
    void cfoo() const; // const-qualified non-static member function  
    void rfoo() &; // ref-qualified non-static member function  
    static void bar(); // static member function  
    Foo(); // Constructor  
    ~Foo(); // Destructor  
    bool operator==(const Foo& f); // Overloaded operator ==  
};
```



Accessing Members

Given the following code:

```
struct C {  
    int i;  
    static int si;  
};  
C o; // o is variable of type C  
C* p = &o; // p is pointer to o
```

the members of the object can be accessed as follows:

- non-static and static member variables and functions can be accessed with the *member-of* operator: `o.i`, `o.si`
- As a shorthand, instead of writing `(*p).i`, it is possible to write `p->i`
- Static member variables and functions can also be accessed with the *scope resolution* operator: `C::si`



Writing Member Functions

- In a non-static member function members can be accessed implicitly without using the member-of operator (preferred)
- Every non-static member function has the implicit parameter `this`
- In member functions without qualifiers and ref-qualified ones `this` has the type `C*`
- In const-qualified or const-ref-qualified member functions `this` has the type `const C*`

```
struct C {  
    int i;  
    int foo() {  
        this->i; // Explicit member access, this has type C*  
        return i; // Implicit member access  
    }  
    int foo() const { return this->i; /* this has type const C* */ }  
    int bar() & { return i; /* this (implicit) has type C* */ }  
    int bar() const& { return this->i; /* this has type const C* */ }  
};
```


Out-of-line Definitions

- Just like regular functions member functions can have separate declarations and definitions
- A member function that is defined in the class body is said to have an *inline definition*
- A member function that is defined outside of the class body is said to have an *out-of-line definition*
- Member functions with inline definitions implicitly have the `inline` specifier
- Out-of-line definitions must have the same qualifiers as their declaration

```
struct Foo {  
    void foo1() { /* ... */ } // Inline definition  
    void foo2();  
    void foo_const() const;  
    static void foo_static();  
};  
// Out-of-line definitions  
void Foo::foo2() { /* ... */ }  
void Foo::foo_const() const { /* ... */ }  
void Foo::foo_static() { /* ... */ }
```



Forward Declarations (1)

Classes can be *forward-declared*

- Syntax: *class-keyword name ;*
- Declares a class type which will be defined later in the scope
- The class name has *incomplete type* until it is defined
- The forward-declared class name may still be used in some situations (more details next)

Use Cases

- Allows classes to refer to each other
- Can reduce compilation time (significantly) by avoiding transitive includes of an expensive-to-compile header
- Commonly used in header files

Forward Declarations (2)

Example

foo.hpp

```
class A;  
class ClassFromExpensiveHeader;  
  
class B {  
    ClassFromExpensiveHeader* member;  
  
    void foo(A& a);  
};  
class A {  
    void foo(B& b);  
};
```

foo.cpp

```
#include "expensive_header.hpp"  
  
/* implementation */
```



Incomplete Types

A forward-declared class type is *incomplete* until it is defined

- In general, no operations that require the size and layout of a type to be known can be performed on an incomplete type
 - E.g. pointer arithmetics on a pointer to an incomplete type
 - E.g. Definition or call (but not declaration) of a function with incomplete return or argument type
- However, some declarations can involve incomplete types
 - E.g. pointer declarations to incomplete types
 - E.g. member function declarations with incomplete parameter types
- For details: See the reference documentation



Constructors

- Constructors are special functions that are called when an object is *initialized*
- Constructors have no return type, no const- or ref-qualifiers, and their name is equal to the class name
- The definition of a constructor can have an *initializer list*
- Constructors can have arguments, a constructor without arguments is called *default constructor*
- Constructors are sometimes implicitly defined by the compiler

```
struct Foo {  
    Foo() {  
        std::cout << "Hello\n";  
    }  
};
```

```
struct Foo {  
    int a;  
    Bar b;  
    // Default constructor is  
    // implicitly defined, does  
    // nothing with a, calls  
    // default constructor of b  
};
```

Initializer List

- The initializer list specifies how member variables are initialized before the body of the constructor is executed
- Other constructors can be called in the initializer list
- Members should be initialized in the order of their definition
- Members are initialized to their default value if not specified in the list
- `const` member variables can only be initialized in the initializer list

```
struct Foo {
    int a = 123; float b; const char c;
    // default constructor initializes a (to 123), b, and c
    Foo() : b(2.5), c(7) {}
    // initializes a and b to the given values
    Foo(int a, float b, char c) : a(a), b(b), c(c) {}
    Foo(float f) : Foo() {
        // First the default constructor is called, then the body
        // of this constructor is executed
        b *= f;
    }
};
```



Initializing Objects

- When an object of class type is initialized, an appropriate constructor is executed
- Arguments given in the initialization are passed to the constructor
- C++ has several types of initialization that are very similar but unfortunately have subtle differences:
 - *default initialization* (`Foo f;`)
 - *value initialization* (`Foo f{};` and `Foo()`)
 - *direct initialization* (`Foo f(1, 2, 3);`)
 - *list initialization* (`Foo f{1, 2, 3};`)
 - *copy initialization* (`Foo f = g;`)
- Simplified syntax: `class-type identifier(arguments);` or `class-type identifier{arguments};`



Converting and Explicit Constructors

- Constructors with exactly one argument are treated specially: They are used for *explicit* and *implicit conversions*
- If implicit conversion with such constructors is not desired, the keyword `explicit` can be used to disallow it
- Generally, you should use `explicit` unless you have a good reason not to

```
struct Foo {  
    Foo(int i);  
};  
void print_foo(Foo f);  
// Implicit conversion,  
// calls Foo::Foo(int)  
print_foo(123);  
// Explicit conversion,  
// calls Foo::Foo(int)  
static_cast<Foo>(123);
```

```
struct Bar {  
    explicit Bar(int i);  
};  
void print_bar(Bar f);  
// Implicit conversion,  
// compiler error!  
print_bar(123);  
// Explicit conversion,  
// calls Bar::Bar(int)  
static_cast<Bar>(123);
```




Copy Constructors

- Constructors of a class C that have a single argument of type C& or `const C&` (preferred) are called *copy constructors*
- They are often called implicitly by the compiler whenever it is necessary to copy an object
- The copy constructor is often implicitly defined by the compiler

```
struct Foo {  
    Foo(const Foo& other) { /* ... */ }  
};  
void doFoo(Foo f);  
Foo f;  
Foo g(f); // Call copy constructor explicitly  
doFoo(g); // Copy constructor is called implicitly
```



Destructors

- The destructor is a special function that is called when the lifetime of an object ends
- The destructor has no return type, no arguments, no const- or ref-qualifiers, and its name is `~class-name`
- For objects with automatic storage duration (e.g. local variables) the destructor is called implicitly at the end of the scope in reverse order of their definition

```
Foo a;  
Bar b;  
{  
    Baz c;  
    // c.~Baz() is called;  
}  
// b.~Bar() is called  
// a.~Foo() is called
```

Writing Destructors

- The destructor is a regular function that can contain any code
- Most of the time the destructor is used to explicitly free resources
- Destructors of member variables are called automatically at the end in reverse order

```
struct Foo {  
    Bar a;  
    Bar b;  
    ~Foo() {  
        std::cout << "Bye\n";  
        // b.~Bar() is called  
        // a.~Bar() is called  
    }  
};
```



Member Access Control

- Every member of a class has **public**, **protected**, or **private** access
- When the class is defined with **class**, the default access is **private**
- When the class is defined with **struct**, the default access is **public**
- **public** members can be accessed by everyone, **protected** members only by the class itself and its subclasses, **private** members only by the class itself

```
class Foo {  
    int a; // a is private  
    public:  
    // All following declarations are public  
    int b;  
    int getA() const { return a; }  
    protected:  
    // All following declarations are protected  
    int c;  
    public:  
    // All following declarations are public  
    static int getX() { return 123; }  
};
```



Friend Declarations (1)

A class body can contain *friend declarations*

- A friend declaration grants a function or another class access to the private and protected members of the class which contains the declaration
- Syntax: `friend function-declaration ;`
 - Declares a function as a friend of the class
- Syntax: `friend function-definition ;`
 - Defines a non-member function and declares it as a friend of the class
- Syntax: `friend class-specifier ;`
 - Declares another class as a friend of this class

Notes

- Friendship is non-transitive and cannot be inherited
- Access specifiers have no influence on friend declarations (i.e. they can appear in `private:` or `public:` sections)

Friend Declarations (2)

Example

```
class A {
    int a;
    friend class B;
    friend void foo(A&);
};
class B {
    friend class C;
    void bar(A& a) {
        a.a = 42; // OK
    }
};
class C {
    void foo(A& a) {
        a.a = 42; // ERROR
    }
};
void foo(A& a) {
    a.a = 42; // OK
}
```

Nested Types

- For nested types classes behave just like a namespace
- Nested types are accessed with the scope resolution operator `::`
- Nested types are **friends** of their parent

```
struct A {  
    struct B {  
        int getI(const A& a) {  
            return a.i; // OK, B is friend of A  
        }  
    };  
    private:  
    int i;  
};  
A::B b; // reference nested type B of class A
```

Constness of Member Variables

- Accessing a member variable through a *non-const lvalue* yields a *non-const lvalue* if the member is non-const and a *const lvalue* otherwise
- Accessing a member variable through a *const lvalue* yields a *const lvalue*
- Exception: Member variables declared with `mutable` yield a *non-const lvalue* even when accessed through a *const lvalue*

```
struct Foo {  
    int i;  
    const int c;  
    mutable int m;  
}  
Foo& foo = /* ... */;  
const Foo& cfoo = /* ... */;
```

Expression	Value Category
<code>foo.i</code>	non-const lvalue
<code>foo.c</code>	const lvalue
<code>foo.m</code>	non-const lvalue
<code>cfoo.i</code>	const lvalue
<code>cfoo.c</code>	const lvalue
<code>cfoo.m</code>	non-const lvalue

Constness and Member Functions

- The value category through which a non-static member function is accessed is taken into account for overload resolution
- For *non-const lvalues* non-const overloads are preferred over const ones
- For *const lvalues* only const-(ref-)qualified functions are selected

```

struct Foo {
    int getA() { return 1; }
    int getA() const { return 2; }
    int getB() & { return getA(); }
    int getB() const& { return getA(); }
    int getC() const { return getA(); }
    int getD() { return 3; }
};
Foo& foo = /* ... */;
const Foo& cfoo = /* ... */;

```

Expression	Value
foo.getA()	1
foo.getB()	1
foo.getC()	2
foo.getD()	3
cfoo.getA()	2
cfoo.getB()	2
cfoo.getC()	2
cfoo.getD()	<i>error</i>



Casting and CV-qualifiers

- When using `static_cast`, `reinterpret_cast`, or `dynamic_cast`, cv-qualifiers cannot be “casted away”
- `const_cast` must be used instead
- Syntax: `const_cast < new_type > (expression)`
- `new_type` may be a pointer or reference to a class type
- `expression` and `new_type` must have same type ignoring their cv-qualifiers
- The result of `const_cast` is a value of type `new_type`
- Modifying a const object through a non-const access path is undefined behavior!

```
struct Foo {
    int a;
};
const Foo f{123};
Foo& fref = const_cast<Foo&>(f); // OK, cast is allowed
int b = fref.a; // OK, accessing value is allowed
fref.a = 42; // undefined behavior
```

Use Cases for `const_cast`

Most common use case of `const_cast`: Avoid code duplication in member function overloads.

- A class may contain a const and non-const overload of the same function with identical code
- Should only be used when absolutely necessary (i.e. not for simple overloads)

```
class A {
    int* numbers;
    int& foo() {
        int i = /* ... */;
        // do some incredibly complicated computation to
        // get a value for i
        return numbers[i]
    }
    const int& foo() const {
        // OK as long as foo() does not modify the object
        return const_cast<A*>(*this).foo();
    }
};
```



Operator Overloading

- Classes can have special member functions to overload built-in operators like +, ==, etc.
- Many overloaded operators can also be written as non-member functions
- Syntax: *return-type operator op (arguments)*
- Overloaded operator functions are selected with the regular overload resolution
- Overloaded operators are not required to have meaningful semantics
- Almost all operators can be overloaded, exceptions are: :: (scope resolution), . (member access), .* (member pointer access), ?: (ternary operator)
- This includes “unusual” operators like: = (assignment), () (call), * (dereference), & (address-of), , (comma)



Arithmetic Operators

The expression $lhs\ op\ rhs$ is mostly equivalent to $lhs.operator\ op(rhs)$ or $operator\ op(lhs, rhs)$ for binary operators.

- As calls to overloaded operators are treated like regular function calls, the overloaded versions of `||` and `&&` lose their special behaviors
- Should be `const` and take `const` references
- Usually return a value and not a reference
- The unary `+` and `-` operators can be overloaded as well

```
struct Int {
    int i;
    Int operator+(const Int& other) const { return Int{i + other.i}; }
    Int operator-() const { return Int{-i}; };
};
Int operator*(const Int& a, const Int& b) { return Int{a.i * b.i}; }

Int a{123}; Int b{456};

a + b; /* is equivalent to */ a.operator+(b);
a * b; /* is equivalent to */ operator*(a, b);
-a; /* is equivalent to */ a.operator-();
```



Comparison Operators

All binary comparison operators (<, <=, >, >=, ==, !=, <=>) can be overloaded.

- Should be const and take const references
- Return `bool`, except for <=> (see next slide)
- If only `operator<=>` is implemented, <, <=, >, and >= work as well
- `operator==` must be implemented separately
- If `operator==` is implemented, != works as well

```
struct Int {
    int i;
    std::strong_ordering operator<=>(const Int& a) const {
        return i <=> a.i;
    }
    bool operator==(const Int& a) const { return i == a.i; }
};
Int a{123}; Int b{456};
a < b; /* is equivalent to */ (a.operator<=>(b)) < 0;
a == b; /* is equivalent to */ a.operator==(b);
```



Three-Way Comparison (1)

The overloaded `operator<=>` should return one of the following three types from `<compare>`: `std::partial_ordering`, `std::weak_ordering`, `std::strong_ordering`.

- When comparing two values `a` and `b` with `ord = (a <=> b)`, then `ord` has one of the three types and can be compared to 0:
- `ord == 0` \Leftrightarrow `a == b`
- `ord < 0` \Leftrightarrow `a < b`
- `ord > 0` \Leftrightarrow `a > b`
- `std::strong_ordering` can be converted to `std::weak_ordering` and `std::partial_ordering`
- `std::weak_ordering` can be converted to `std::partial_ordering`

Three-Way Comparison (2)

`std::partial_ordering` should be used when two values can potentially be unordered, i.e. $a \leq b$ and $a \geq b$ could be false.

Possible values:

- `std::partial_ordering::less`
- `std::partial_ordering::equivalent`
- `std::partial_ordering::greater`
- `std::partial_ordering::unordered`

Three-Way Comparison (3)

`std::weak_ordering` or `std::strong_ordering` should be used when two values are always ordered (i.e. we have *total order*).

Possible values:

- `std::weak_ordering::less`
- `std::weak_ordering::equivalent`
- `std::weak_ordering::greater`
- `std::strong_ordering::less`
- `std::strong_ordering::equivalent`
- `std::strong_ordering::greater`
- With `std::strong_ordering` equal values must also be “indistinguishable”, i.e. behave the same in all aspects



Increment and Decrement Operators

Overloaded pre- and post-increment and -decrement operators are distinguished by an (unused) `int` argument.

- C& `operator++()`; C& `operator--()`; overloads the pre-increment or -decrement operator, usually modifies the object and then returns `*this`
- C `operator++(int)`; C `operator--(int)`; overloads the post-increment or -decrement operator, usually copies the object before modifying it and then returns the unmodified copy

```
struct Int {
    int i;
    Int& operator++() { ++i; return *this; }
    Int operator--(int) { Int copy{*this}; --i; return copy; }
};
Int a{123};
++a; // a.i is now 124
a++; // ERROR: post-increment is not overloaded
Int b = a--; // b.i is 124, a.i is 123
--b; // ERROR: pre-decrement is not overloaded
```



Subscript Operator

Classes that behave like containers or pointers usually override the *subscript operator* `[]`.

- `a[b]` is equivalent to `a.operator[](b)`
- Type of `b` can be anything, for array-like containers it is usually `size_t`

```
struct Foo { /* ... */ };
struct FooContainer {
    Foo* fooArray;
    Foo& operator[](size_t n) { return fooArray[n]; }
    const Foo& operator[](size_t n) const { return fooArray[n]; }
};
```



Dereference Operators

Classes that behave like pointers usually override the operators `*` (dereference) and `->` (member of pointer).

- `operator*`() usually returns a reference
- `operator->`() should return a pointer or an object that itself has an overloaded `->` operator

```
struct Foo { /* ... */ };  
struct FooPtr {  
    Foo* ptr;  
    Foo& operator*() { return *ptr; }  
    const Foo& operator*() const { return *ptr; }  
    Foo* operator->() { return ptr; }  
    const Foo* operator->() const { return ptr; }  
};
```



Assignment Operators

- The simple assignment operator is often used together with the copy constructor and should have the same semantics
- All assignment operators usually return `*this`

```
struct Int {  
    int i;  
    Foo& operator=(const Foo& other) { i = other.i; return *this; }  
    Foo& operator+=(const Foo& other) { i += other.i; return *this; }  
};  
Foo a{123};  
a = Foo{456}; // a.i is now 456  
a += Foo{1}; // a.i is now 457
```



Conversion Operators

A class C can use converting constructors to convert values of other types to type C. Similarly, *conversion operators* can be used to convert objects of type C to other types.

Syntax: `operator type ()`

- Conversion operators have the implicit return type *type*
- They are usually declared as `const`
- The `explicit` keyword can be used to prevent implicit conversions
- Explicit conversions are done with `static_cast`
- `operator bool()` is usually overloaded to be able to use objects in an `if` statement

```
struct Int {  
    int i;  
    operator int() const {  
        return i;  
    }  
};  
Int a{123};  
int x = a; // OK, x is 123
```

```
struct Float {  
    float f;  
    explicit operator float() const {  
        return f;  
    }  
};  
Float b{1.0};  
float y = b; // ERROR, implicit conversion  
float y = static_cast<float>(b); // OK
```



Argument-Dependent Lookup

- Overloaded operators are usually defined in the same namespace as the type of one of their arguments
- Regular unqualified lookup would not allow the following example to compile
- To fix this, unqualified names of functions are also looked up in the *namespaces of all arguments*
- This is called *Argument Dependent Lookup (ADL)*

```
namespace A { class X {}; X operator+(const X&, const X&); }
int main() {
    A::X x, y;
    operator+(x, y); // Need operator+ from namespace A
    A::operator+(x, y); // OK
    x + y; // How to specify namespace here?
           // -> ADL finds A::operator+()
}
```



Defaulted Member Functions

- Most of the time the implementation of default constructors, copy constructors, copy assignment operators, and destructors is trivial
- To let the compiler generate the trivial implementation automatically, `= default;` can be used instead of a function body

```
struct Foo {  
    Bar b;  
    Foo() = default; /* equivalent to: */ Foo() {}  
    ~Foo() = default; /* equivalent to: */ ~Foo() {}  
  
    Foo(const Foo& f) = default;  
    /* equivalent to: */  
    Foo(const Foo& f) : b(f.b) {}  
  
    Foo& operator=(const Foo& f) = default;  
    /* equivalent to: */  
    Foo& operator=(const Foo& f) {  
        b = f.b; return *this;  
    }  
};
```




Defaulted Comparison Operators

All comparison operators can be defaulted.

- Defaulted comparison operators must return `bool`, except `<=>`
- Defaulted `operator==` compares each member for equality, members must define `operator==`
- Defaulted `operator<=>` lexicographically compares members by using `<=>`, members must define `operator<=>`
- Defaulting `operator<=>` also defaults `operator==`
- Defaulted `<`, `<=`, `>`, or `>=` use `operator<=>`

```
struct Int128 {
    int64_t x; int64_t y;
    std::strong_ordering operator<=>(const Int&) const = default;
};
Int128 a{0, 123}; Int128 b{1, 0};
a < b; // true
a == b; // false
a <=> b; // std::strong_ordering::less
```



Deleted Member Functions

- Sometimes, implicitly generated constructors or assignment operators are not wanted
- Writing `= delete;` instead of a function body explicitly forbids implicit definitions
- In other cases the compiler implicitly deletes a constructor in which case writing `= default;` enables it again

```
struct Foo {  
    Foo(const Foo&) = delete;  
};  
Foo f; // Default constructor is defined implicitly  
Foo g(f); // ERROR: copy constructor is deleted
```

Other User-Defined Types



Unions

- In addition to regular classes declared with `class` or `struct`, there is another special class type declared with `union`
- In a union only one member may be “active”, all members use the same storage
- Size of the union is equal to size of largest member
- Alignment of the union is equal to largest alignment among members
- Strict aliasing rule still applies with unions!
- Most of the time there are better alternatives to unions, e.g. `std::array<char, N>` or `std::variant`

```
union Foo {  
    int a;  
    double b;  
};  
sizeof(Foo) == 8;  
alignof(Foo) == 8;
```

```
Foo f; // No member is active  
f.a = 1; // a is active  
std::cout << f.b; // Undefined behavior!  
f.b = 12.34; // Now, b is active  
std::cout << f.b; // OK
```



Enums

- C++ also has user-defined enumeration types
- Typically used like integral types with a restricted range of values
- Also used to be able to use descriptive names instead of “magic” integer values
- Syntax: *enum-key name { enum-list };*
- *enum-key* can be **enum**, **enum class**, or **enum struct**
- *enum-list* consists of comma-separated entries with the following syntax:
name [= value]
- When *value* is not specified, it is automatically chosen starting from 0

```
enum Color {  
    Red, // Red == 0  
    Blue, // Blue == 1  
    Green, // Green == 2  
    White = 10,  
    Black, // Black == 11  
    Transparent = White // Transparent == 10  
};
```

Using Enum Values

- Names from the enum list can be accessed with the scope resolution operator
- When `enum` is used as keyword, names are also introduced in the enclosing namespace
- Enums declared with `enum` can be converted implicitly to `int`
- Enums can be converted to integers and vice versa with `static_cast`
- `enum class` and `enum struct` are equivalent
- Guideline: Use `enum class` unless you have a good reason not to

```
Color::Red; // Access with scope resolution operator
Blue; // Access from enclosing namespace
int i = Color::Green; // i == 2, implicit conversion
int j = static_cast<int>(Color::White); // j == 10
Color c = static_cast<Color>(11); // c == Color::Black
```



Type Aliases

- Names of types that are nested deeply in multiple namespaces or classes can become very long
- Sometimes it is useful to declare a nested type that refers to another, existing type
- For this *type aliases* can be used
- Syntax: `using name = type;`
- *name* is the name of the alias, *type* must be an existing type
- For compatibility with C type aliases can also be defined with `typedef` with a different syntax but this should never be used in modern C++ code

```
namespace A::B::C { struct D { struct E {}; }; }  
using E = A::B::C::D::E;  
E e; // e has type A::B::C::D::E  
struct MyContainer {  
    using value_type = int;  
};  
MyContainer::value_type i = 123; // i is an int
```

Common Type Aliases

In C++ the following aliases are defined in the `std` namespace and are commonly used:

`intN_t`: Integer types with exactly N bits, usually defined for 8, 16, 32, and 64 bits

`uintN_t`: Similar to `intN_t` but unsigned

`size_t`: Used by the standard library containers everywhere a size or index is needed, also result type of `sizeof` and `alignof`

`uintptr_t`: An integer type that is guaranteed to be able to hold all possible values that result from a `reinterpret_cast` from any pointer

`intptr_t`: Similar to `uintptr_t` but signed

`ptrdiff_t`: Result type of expressions that subtract two pointers

`max_align_t`: Type which has alignment as least as large as all other scalar types