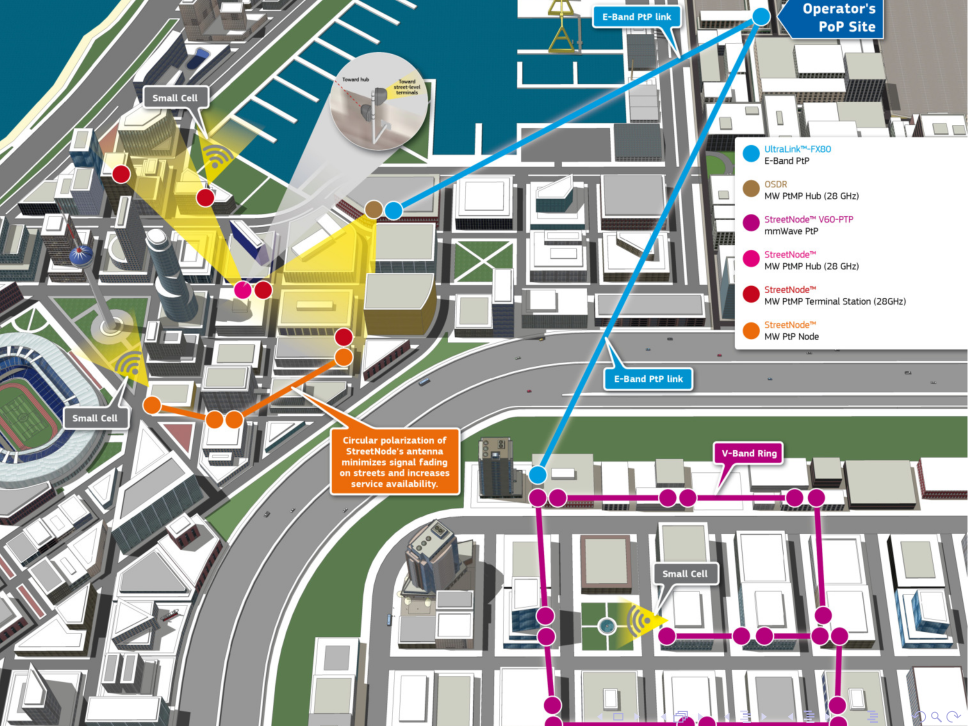
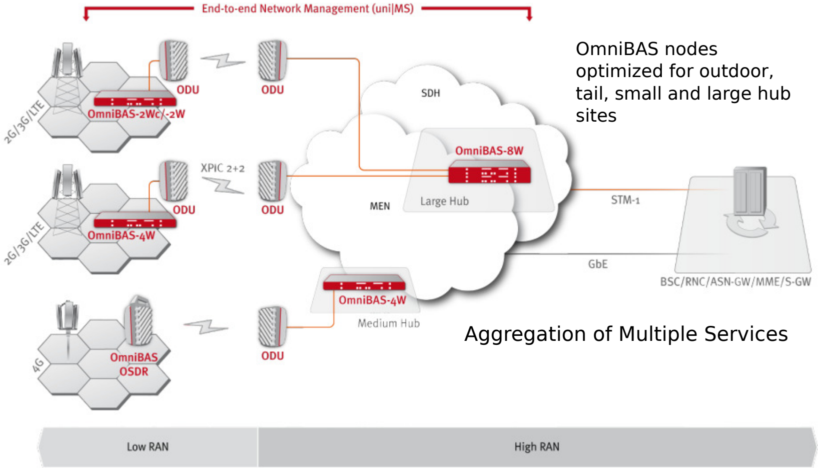
Agenda

1. What we build
2. Tools and Practices
3. C++ Language
4. STL Library
5. Conclusions

Outline


Operator's PoP Site
E-Band PtP link
Small Cell
Toward hub
Toward street-level terminals
UltraLink™-FX80
E-Band PtP
OSDR
MW PtMP Hub (28 GHz)
StreetNode™ V60-PTP
mmWave PtP
StreetNode™
MW PtMP Hub (28 GHz)
StreetNode™
MW PtMP Terminal Station (28GHz)
StreetNode™
MW PtP Node
E-Band PtP link
Small Cell
Circular polarization of StreetNode's antenna minimizes signal fading on streets and increases service availability.
V-Band Ring
Small Cell

Product Portfolio
Point to Point

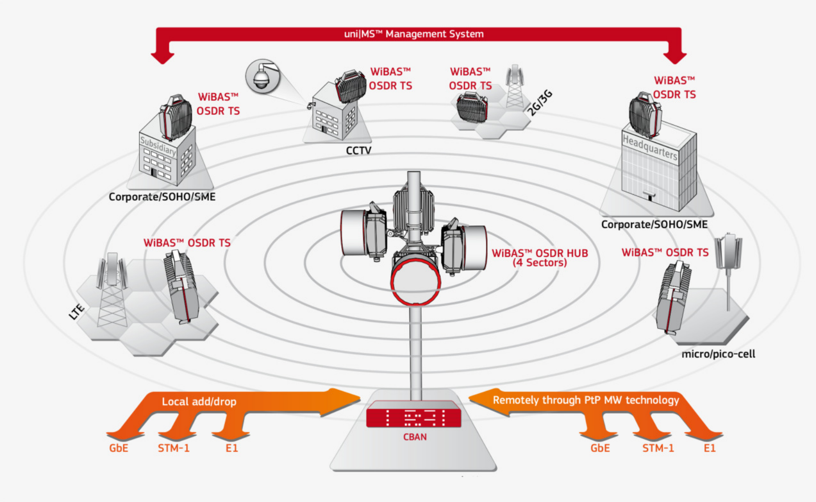


OmniBAS nodes optimized for outdoor, tail, small and large hub sites

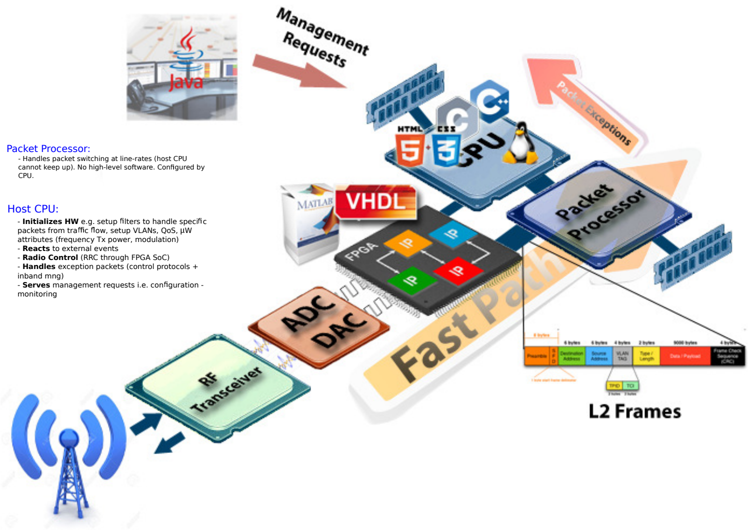
Aggregation of Multiple Services

Product Portfolio

Point to Multi-Point



Product Portfolio
Element Internals



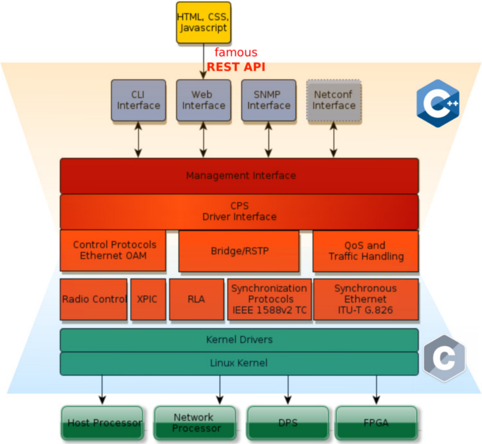
Packet Processor:

- Handles packet switching at line-rates (host CPU cannot keep up). No high-level software. Configured by CPU.

Host CPU:

- **Initializes HW** e.g. setup filters to handle specific packets from traffic flow, setup VLANs, QoS, µW attributes (frequency Tx power, modulation)
- **Reacts** to external events
- **Radio Control** (RRC through FPGA SoC)
- **Handles** exception packets (control protocols + inband mng)
- **Serves** management requests i.e. configuration - monitoring

How C++ is used inhouse

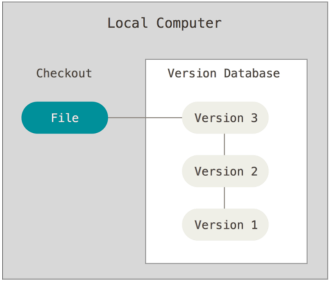


Outline

Source Version Control

What is version control?

- system that records changes to a file or set of files over time so that you can recall specific versions later
- can be centralized or distributed
- changes are usually identified by a number or letter code, termed the "revision number", "revision level", or simply "revision"
- benefits:
  - backup and restore
  - track changes
  - collaborate with team



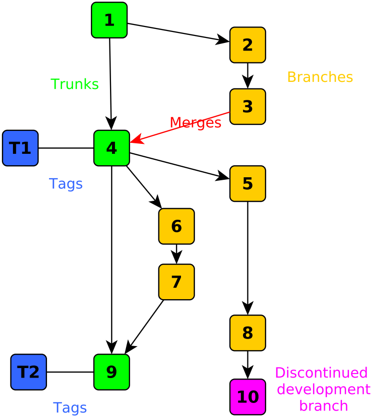
Source Version Control

Main Concepts

- Repository
- Trunk or Mainline
- Branch
- Merge
- Tag

Main Actions

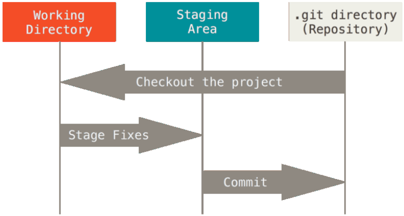
- Clone
- Add
- Commit - Check in
- Check out
- Update/Sync
- Revert
- Resolve Conflict



Source Version Control
git

- fast, flexible, but challenging *distributed* version control system
- stream of snapshots (*commits* with SHA-1 hash ids)
- working directory, staging area, repository

```
# add to staging area
$ git add foo.txt

# commit staged files to repository
$ git commit -m "message"

# undo local (unstaged) changes
$ git checkout foo.txt

# unstage staged changes
$ git reset HEAD foo.txt

# show commit logs (history)
$ git log foo.txt
```



Source Version Control

git

- git has remote and local repository
- add and commit often, push only when ready!

```
# clone from remote repository
$ git clone git@github:/username/cppworkshop.git

# synchronize master branch with remote repository (preferred approach)
$ git fetch -p
$ git merge origin/master
# synchronize master branch with remote repository (quick approach)
$ git pull

# push master branch to remote repository
$ git push origin master

# branches
$ git branch -a        # list remote and local branches
$ git checkout master  # switch working directory to branch 'master'
$ git branch dev       # create a new branch named 'dev'
$ git merge dev        # merge branch 'dev' into working branch
$ git branch -d dev    # delete branch 'dev'
```

Build System

Build automation

involves scripting or automating the process of compiling computer source code into binary code.

Popular build systems:

Make-based tools: GNU Make, nmake, ...

Non-Make based tools: apache ant, MS Build, bazel, b2 or bjam, sCons, Ninja

Build script generation tools: configure, CMake(our choice), autotools, qmake

Continuous integration tools: Jenkins, Travis CI(our choice), Deploy Bot

Build System
CMake

CMake

is cross-platform free and open-source software for managing the build process of software using a compiler-independent method. It is used in conjunction with native build environments such as make, Apple's Xcode, and Microsoft Visual Studio.

Build Process:

1. Standard build files are created from configuration files. Each build project contains a `CMakeLists.txt` file in every directory that controls the build process.
2. The platform's native build tools are used for the actual building.

Build System

CMake

```
mkdir build && cd build
INSTALLDIR=. INSTALLDIR_BIN="./bin" INSTALLDIR_LIB="./lib" \
BINDIR="./bin" LIBDIR="./lib" cmake ..  # STAGE 1
-- The CXX compiler identification is GNU 5.3.1
# ...
-- Performing Test COMPILER_SUPPORTS_CXX14
-- Performing Test COMPILER_SUPPORTS_CXX14 - Success
# ...
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Found Threads: TRUE
-- Configuring done
-- Generating done
-- Build files have been written to: /test/cppworkshop/build
make all                                # STAGE 2
Scanning dependencies of target future
[ 25%] Building CXX object src/CMakeFiles/future.dir/future.cpp.o
[ 50%] Linking CXX executable ../../bin/future
[ 50%] Built target future
Scanning dependencies of target memerr
[ 75%] Building CXX object src/CMakeFiles/memerr.dir/memerr.cpp.o
[100%] Linking CXX executable ../../bin/memerr
[100%] Built target memerr
make test                               # Testing STAGE
  Running tests...
100% tests passed, 0 tests failed out of 2
Total Test time (real) =   0.01 sec
```

Continuous Integration

In software engineering, continuous integration (CI) is the practice of merging all developer working copies to a shared mainline several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early.

- The goal is to have the software always in working state.
- Is a practice, not a tool!
- Increases visibility, which enables greater communication.
- Easy to find and fix bugs.
- Requires:
  - To check in regularly.
  - To have automated test suite.
  - To have short build and test processes.
  - To have a common development environment.

Continuous Integration

Testing

Software testing

involves the execution of a software component or system component to evaluate one or more properties of interest:

- meets the requirements that guided its design and development
- responds correctly to all kinds of inputs
- performs its functions within an acceptable time
- is sufficiently usable
- can be installed and run in its intended environments
- achieves the general result its stakeholders desire

Continuous Integration
Testing Levels

Unit testing is tests that verify the functionality of a specific section of code, usually at the function level.

Integration testing is any type of software testing that seeks to verify the interfaces between components against a software design.

System Testing or end-to-end testing, tests a completely integrated system to verify that the system meets its requirements.

Continuous Integration

Unit Testing

What is unit testing?

- lowest level of testing performed
- individual units of software are tested
- the purpose is to validate that each unit of the software performs as designed

Continuous Integration

Unit Testing

What is unit testing?

- lowest level of testing performed
- individual units of software are tested
- the purpose is to validate that each unit of the software performs as designed

Advantages

- each part tested individually
- all components tested at least once
- errors picked up earlier
- scope is smaller, easier to fix errors

Continuous Integration
Unit Testing

What is unit testing?

- lowest level of testing performed
- individual units of software are tested
- the purpose is to validate that each unit of the software performs as designed

Advantages

- each part tested individually
- all components tested at least once
- errors picked up earlier
- scope is smaller, easier to fix errors

Unit tests ideals

- isolatable
- repeatable
- automatable
- easy to write and read

Language Tools

Sanitizers

Address Sanitizer

is a fast memory error detector. It consists of a compiler instrumentation module and a run-time library.

The tool can detect the following types of bugs:

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return
- Use-after-scope
- Double-free, invalid free
- Memory leaks

Typical slowdown introduced by AddressSanitizer is 2x. Supported from `clang`, `gcc` by compiling and linking your program with `-fsanitize=address` flag.

Language Tools

Pointer Pitfalls

- use uninitialized pointer:

```
X* x; use(*x);
```

- out-of-bounds access:

```
X* x = new X[10]; use(x[10]);
```

- use after free:

```
X* x = new X; delete x; use(*x);
```

- dangling pointers:

```
X *x = new X, *p = x; delete x; use(*p);
```

- memory leaks:

```
X *x = new X; if (use(*x)) return; delete x;
```

- many, many more!

Language Tools

Valgrind

Valgrind

is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail.

- Several tools are included by default:
  - Memcheck (rock star)
  - Cachegrind
  - Helgrind
  - more...
- Simulates every single instruction (including libraries, suppressions)
- Need to start the program with Valgrind attached
- Makes program run 10-50 times slower while in use

Language Tools

Static Analysis

Source code analysis

is the analysis of computer software that is performed without actually executing programs. In most cases the analysis is performed on some version of the source code, and in the other cases, some form of the object code.

Set of analyzing techniques:

- Data-flow Analysis (DFA) is a technique for gathering information about the possible set of values calculated at various points in a computer program.
- Symbolic Execution is a means of analyzing a program to determine what inputs cause each part of a program to execute.
- Dependence Analysis produces execution-order constraints between statements/instructions.

Language Tools
Static Analysis

The analysis can identify:

- Memory leaks
- Dangling pointers
- Uninitialized variables
- Buffer overflow
- Concurrency Issues deadlock, race conditions
- Performance bottlenecks
- API usage errors
- Copy/Paste errors
- Integer overflows, integer handling issues
- Insecure data handling
- Security best practices violations

Hands-on Example

memerr.cpp: memory errors

```
# clone (if you haven't already)
git clone git@github.com:yourname/cppworkshop.git
git clone https://github.com/apmanol/cppworkshop.git
cd cppworkshop

# sync first (if you have a forked github)
git remote add upstream https://github.com/apmanol/cppworkshop.git
git fetch upstream
git checkout master
git merge upstream/master

# build
make distclean
make debug=1

# run
./bin/memerr
```

Outline

C++ in Two Lines

Direct map to hardware

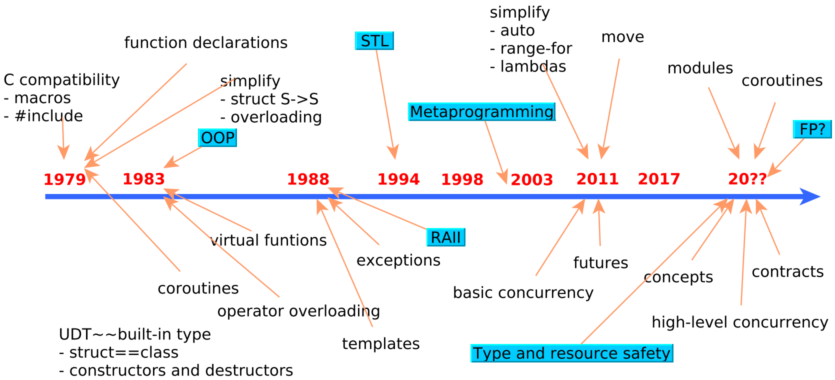
- of instructions and fundamental data types
- initially from C
- future: use novel hardware better (caches, multicores, GPUs, FPGAs, SIMD, ...)

Zero-overhead abstraction

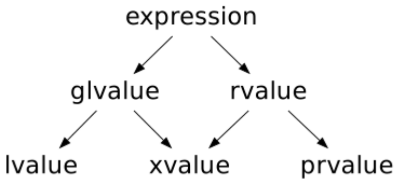
- classes, inheritance, generic programming, ...
- initially from Simula (where it wasn't zero-overhead)
- future: type and resource safety, concepts, modules, concurrency, ...

Key Language Features

Major Design Decisions



Expression Value Categories




```
int    prvalue(); // return ‘‘pure’’ rvalue
int&   lvalue();  // return lvalue
int&& xvalue();  // return ‘‘expiring’’ rvalue
```

Lvalues vs Rvalues

lvalue (“left value”)

- expression that has identity
- cannot be moved from

rvalue (“right value”)

- either a prvalue (“pure”) or an xvalue (“eXpiring”)
- can be moved from
- may or may not have identity

lvalue vs rvalue heuristic

- if you can take the address of an expression, it’s usually an lvalue
- if you can’t, it’s usually an rvalue

Rvalues

rvalues are

- temporaries
- objects without name
- objects whose addresses cannot be determined

lvalues

```
int i = 42;
i = 43;
int* p = &i;
int& foo();
foo() = 42;
int* p1 = &foo();
```

rvalues

```
int j = 0;
j = 42;
int foobar();
j = foobar();
foobar() = 12; // ERROR
int* p2 = &foobar(); // ERROR
```

References

- a reference is an alias for an object
- always refers to the object to which it was initialized
- there is no "null reference" (although there is dangling reference)
- *lvalue references* (X&) refer to objects whose value we want to change (only bind to lvalues)
- *const references* (`const X&`) refer to objects whose value we do not want to change (bind to both lvalues & rvalues)
- *rvalue references* (X&&) refer to objects whose value we do not need to preserve after we have used it (only bind to rvalues)

Template Type Deduction

```
template <typename T> void f(ParamType p);
f(expr);
```

- deduce T and ParamType from expr
- three cases for ParamType:
  1. pointer or reference, but not a universal reference
  2. universal reference
  3. neither a pointer nor a reference

Template Type Deduction: Reference/Pointer

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

1. if expr's type is a reference, then ignore the reference part
2. match expr's type against ParamType to determine T

Template Type Deduction: Reference/Pointer

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

1. if expr's type is a reference, then ignore the reference part
2. match expr's type against ParamType to determine T

```
int x = 42;
f(x);
```

Template Type Deduction: Reference/Pointer

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

1. if expr's type is a reference, then ignore the reference part
2. match expr's type against ParamType to determine T

```
int x = 42;
f(x); // T: int, ParamType: int&
```

Template Type Deduction: Reference/Pointer

```cpp
template <typename T> void f(T& p);
f(expr);
```

1. if expr's type is a reference, then ignore the reference part
2. match expr's type against ParamType to determine T

```cpp
int x = 42;
f(x); // T: int, ParamType: int&

const int cx = x;
f(cx);
```

Template Type Deduction: Reference/Pointer

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

1. if expr's type is a reference, then ignore the reference part
2. match expr's type against ParamType to determine T

```
int x = 42;
f(x); // T: int, ParamType: int&

const int cx = x;
f(cx); // T: const int, ParamType: const int&
```

Template Type Deduction: Reference/Pointer

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

1. if expr's type is a reference, then ignore the reference part
2. match expr's type against ParamType to determine T

```
int x = 42;
f(x); // T: int, ParamType: int&

const int cx = x;
f(cx); // T: const int, ParamType: const int&

const int& rx = x;
f(rx);
```

Template Type Deduction: Reference/Pointer

```cpp
template <typename T> void f(T& p);
f(expr);
```

1. if expr's type is a reference, then ignore the reference part
2. match expr's type against ParamType to determine T

```cpp
int x = 42;
f(x); // T: int, ParamType: int&

const int cx = x;
f(cx); // T: const int, ParamType: const int&

const int& rx = x;
f(rx); // T: const int, ParamType: const int&
```

Template Type Deduction: Universal Reference

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

1. if expr is an lvalue, both T and ParamType are deduced to be lvalue references
2. if expr is an rvalue, the Reference/Pointer rules apply

Template Type Deduction: Universal Reference

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

1. if expr is an lvalue, both T and ParamType are deduced to be lvalue references
2. if expr is an rvalue, the Reference/Pointer rules apply

```
int x = 42;
f(x);
```

Template Type Deduction: Universal Reference

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

1. if expr is an lvalue, both T and ParamType are deduced to be lvalue references
2. if expr is an rvalue, the Reference/Pointer rules apply

```
int x = 42;
f(x); // T: int&, ParamType: int&
```

Template Type Deduction: Universal Reference

```cpp
template <typename T> void f(T&& p);
f(expr);
```

1. if expr is an lvalue, both T and ParamType are deduced to be lvalue references
2. if expr is an rvalue, the Reference/Pointer rules apply

```cpp
int x = 42;
f(x); // T: int&, ParamType: int&
const int cx = x;
f(cx);
```

Template Type Deduction: Universal Reference

```cpp
template <typename T> void f(T&& p);
f(expr);
```

1. if expr is an lvalue, both T and ParamType are deduced to be lvalue references
2. if expr is an rvalue, the Reference/Pointer rules apply

```cpp
int x = 42;
f(x); // T: int&, ParamType: int&
const int cx = x;
f(cx); // T: const int&, ParamType: const int&
```

Template Type Deduction: Universal Reference

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

1. if expr is an lvalue, both T and ParamType are deduced to be lvalue references
2. if expr is an rvalue, the Reference/Pointer rules apply

```
int x = 42;
f(x); // T: int&, ParamType: int&
const int cx = x;
f(cx); // T: const int&, ParamType: const int&
const int& rx = x;
f(rx);
```

Template Type Deduction: Universal Reference

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

1. if expr is an lvalue, both T and ParamType are deduced to be lvalue references
2. if expr is an rvalue, the Reference/Pointer rules apply

```
int x = 42;
f(x); // T: int&, ParamType: int&
const int cx = x;
f(cx); // T: const int&, ParamType: const int&
const int& rx = x;
f(rx); // T: const int&, ParamType: const int&
```

Template Type Deduction: Universal Reference

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

1. if expr is an lvalue, both T and ParamType are deduced to be lvalue references
2. if expr is an rvalue, the Reference/Pointer rules apply

```
int x = 42;
f(x); // T: int&, ParamType: int&
const int cx = x;
f(cx); // T: const int&, ParamType: const int&
const int& rx = x;
f(rx); // T: const int&, ParamType: const int&
f(42);
```

Template Type Deduction: Universal Reference

```cpp
template <typename T> void f(T&& p);
f(expr);
```

1. if expr is an lvalue, both T and ParamType are deduced to be lvalue references
2. if expr is an rvalue, the Reference/Pointer rules apply

```cpp
int x = 42;
f(x); // T: int&, ParamType: int&
const int cx = x;
f(cx); // T: const int&, ParamType: const int&
const int& rx = x;
f(rx); // T: const int&, ParamType: const int&
f(42); // T: int, ParamType: int&&
```

Template Type Deduction: Not Pointer/Reference

```
template <typename T> void f(T p);
f(expr);
```

1. if expr's type is a reference, ignore the reference part
2. if resulting expr is const/volatile, ignore const/volatile

Template Type Deduction: Not Pointer/Reference

```
template <typename T> void f(T p);
f(expr);
```

1. if expr's type is a reference, ignore the reference part
2. if resulting expr is const/volatile, ignore const/volatile

```
int x = 42;
f(x);
```

Template Type Deduction: Not Pointer/Reference

```
template <typename T> void f(T p);
f(expr);
```

1. if expr's type is a reference, ignore the reference part
2. if resulting expr is const/volatile, ignore const/volatile

```
int x = 42;
f(x); // T: int, ParamType: int
```

Template Type Deduction: Not Pointer/Reference

```
template <typename T> void f(T p);
f(expr);
```

1. if expr's type is a reference, ignore the reference part
2. if resulting expr is const/volatile, ignore const/volatile

```
int x = 42;
f(x); // T: int, ParamType: int

const int cx = x;
f(cx);
```

Template Type Deduction: Not Pointer/Reference

```
template <typename T> void f(T p);
f(expr);
```

1. if expr's type is a reference, ignore the reference part
2. if resulting expr is const/volatile, ignore const/volatile

```
int x = 42;
f(x); // T: int, ParamType: int

const int cx = x;
f(cx); // T: int, ParamType: int
```

Template Type Deduction: Not Pointer/Reference

```
template <typename T> void f(T p);
f(expr);
```

1. if expr's type is a reference, ignore the reference part
2. if resulting expr is const/volatile, ignore const/volatile

```
int x = 42;
f(x); // T: int, ParamType: int

const int cx = x;
f(cx); // T: int, ParamType: int

const int& rx = x;
f(rx);
```

Template Type Deduction: Not Pointer/Reference

```
template <typename T> void f(T p);
f(expr);
```

1. if expr's type is a reference, ignore the reference part
2. if resulting expr is const/volatile, ignore const/volatile

```
int x = 42;
f(x); // T: int, ParamType: int

const int cx = x;
f(cx); // T: int, ParamType: int

const int& rx = x;
f(rx); // T: int, ParamType: int
```

Automatic Type Deduction

auto

Specifies that the type of the variable that is being declared will be automatically deduced from its initializer.

```
auto i = 1.1;     // i: double

// You can also qualify auto with cv-qualifiers,
// pointer(*) and reference(&) declarators. E.g.:

double* pd;
auto x = pd;      // x: double*
auto* y = pd;     // y: double*
int g();
auto x = g();           // x: int
const auto& y = g();    // y: const int&
```

Automatic Type Deduction

Use of auto to deduce the type of a variable

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

// in C++98, we'd have to write

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

Note

old meaning of auto (“this is a local variable”) is now illegal

Automatic Type Deduction

auto type deduction

- same as template type deduction:

```
auto x = expr;          // template <typename T> void f(T);          f(expr);
const auto x = expr;    // template <typename T> void f(const T);    f(expr);
auto& x = expr;         // template <typename T> void f(T&);         f(expr);
const auto& x = expr;   // template <typename T> void f(const T&);   f(expr);
auto&& x = expr;        // template <typename T> void f(T&&);        f(expr);
const auto&& x = expr; // template <typename T> void f(const T&&); f(expr);
```

- initializer lists are an exception:

```
auto x = {11, 23, 9}; // x: std::initializer_list<int>

template <typename T> void f(T param);
f({11, 23, 9}); // error
```

Automatic Type Deduction

auto advantages

- must be initialized, e.g.,

```
int x; // maybe uninitialized, but compiles
auto x; // will not compile
```

- avoid type errors, e.g.,

```
std::vector<int> v;
unsigned sz = v.size(); // should be: std::vector<int>::size_type
```

- avoid type conversions in ranged-for loops & lambda closures, e.g.,

```
std::unordered_map<std::string, int> m;
for (const std::pair<std::string, int>& p : m) { ... };
for (const auto& p : m) { ... }; // std::pair<const std::string, int>
```

- eases refactoring, e.g.,

```
long f();
auto val = f(); // can refactor f()'s return type
```

Automatic Type Deduction

decltype

Inspects the declared type of an entity or queries the type and value category of an expression.

```
int i = 33;
decltype(i) j = i * 2;

struct A { double x; };
const A a{3.14};
decltype(a.x) x2 = a.x; // x2: double
decltype(A::x) x3 = x2; // ditto

template <typename T, typename U>
auto add(T t, U u) -> decltype(t + u); // return type depends on template parameters

auto f = [](int a, int b) -> int { return a * b; };
decltype(f) f2 = f; // the type of a lambda function is unique and unnamed
i = f(2, 2);
j = f2(3, 3);
```

Automatic Type Deduction

Use decltype to verify deduced type from compiler error diagnostics

```
# cat << EOF > type.cpp
template <typename T>
struct incomplete;

int main()
{
  auto var = "Hello World";
  incomplete<decltype(var)> varType;
  return 0;
}
EOF

# g++ -o type -O2 -Wall -g -std=c++11 type.cpp
type.cpp: In function 'int main()':
type.cpp:7:29: error: aggregate 'incomplete<const char*> varType'
has incomplete type and cannot be defined
   incomplete<decltype(var)> varType;
```

Hands-on Example

auto.cpp: type deduction

```
# clone (if you haven't already)
git clone git@github.com:yourname/cppworkshop.git
git clone https://github.com/apmanol/cppworkshop.git
cd cppworkshop

# build
make distclean
make debug=1

# run
./bin/auto
```

Lambda Functions

lambda

Constructs a closure: an unnamed function object capable of capturing variables in scope.

```
[capture-list] (params) -> ret {body}
```

- capture-list: comma-separated captures (identifier, &identifier, =identifier)
- params: list of parameters, as in named functions
- ret: return type
- body: function body

Lambda Functions

```
auto glambda = [](int a, int b) {
  return a < b; };
```

Lambda Functions

```
auto glambda = [](int a, int b) {
  return a < b; };
```

```
// C++98
struct functor
{
  explicit functor(int a) : a_(a) {}
  bool operator()(int x) const
  {
    return x == a_;
  }
 private:
  int a_;
};

int a = 42;
std::count_if(v.begin(), v.end(),
    functor(a));
```

Lambda Functions

```cpp
auto glambda = [](int a, int b) {
  return a < b; };
```

```cpp
// C++98
struct functor
{
  explicit functor(int a) : a_(a) {}
  bool operator()(int x) const
  {
    return x == a_;
  }
 private:
  int a_;
};

int a = 42;
std::count_if(v.begin(), v.end(),
    functor(a));
```

```cpp
// C++11
int a = 42;
std::count_if(v.cbegin(), v.cend(),
    [a](int x){ return x == a; });
```

Lambda Functions

```
auto glambda = [](int a, int b) {
  return a < b; };
```

```
// C++98
struct functor
{
  explicit functor(int a) : a_(a) {}
  bool operator()(int x) const
  {
    return x == a_;
  }
 private:
  int a_;
};

int a = 42;
std::count_if(v.begin(), v.end(),
    functor(a));
```

```
// C++11
int a = 42;
std::count_if(v.cbegin(), v.cend(),
    [a](int x){ return x == a; });


// C++14
std::count_if(v.cbegin(), v.cend(),
    [a](auto x){ return x == a; });
```

Lambda Functions
Captures

- captures apply only to non-static local variables (including parameters); namespace/global variables are always accessible
- `[a, &b]` a is captured by value and b is captured by reference
- `[this]` captures the this pointer by value
- `[&]` captures all local variables by reference
- `[=]` capture all local variables by value
- `[]` captures nothing
- advice: avoid default capture modes (i.e., [=] and [&])

Lambda Functions
Captures

```
void f()
{
  int x = 4;
  int y = 5;
  [&]() {x = 2; y = 2;}();              // f:{x=2, y=2}, lambda:{x=2, y=2}
  [=]() mutable {x = 3; y = 5;}();      // f:{x=2, y=2}, lambda:{x=3, y=5}
  [=, &x]() mutable {x = 7; y = 9;}(); // f:{x=7, y=2}, lambda:{x=7, y=9}
}
```

```
struct S { void f(int i); };
void S::f(int i)
{
  [&, i] {}();     // ok
  [&, &i] {}();    // error: i preceded by & when & is the default
  [=, this] {}(); // error: this when = is the default
  [i, i] {}();     // error: i repeated
}
```

Generic Lambdas

Generic Lambda

C++14 allows lambda function parameters to be declared with the `auto` type specifier

```
auto lambda = [](auto x, auto y) { return x + y; };

std::vector<int> v(10);
std::for_each(v.begin(), v.end(),
              [](auto& i) { ++i; });
std::for_each(v.cbegin(), v.cend(),
              [](auto i) { std::cout << i << " "; });
```

Range-for

- executes a for loop accessing each element of a range
- looks for v.begin() and v.end(), or begin(v) and end(v)

```
std::vector<int> v = {0, 1, 2, 3, 4, 5};
// access by value, i: int
for (auto i: v)
  std::cout << i << '\n';
// access by reference, i: int&
for (auto& i: v)
  ++i;

std::vector<std::string> vs {"C++98", "C++11", "C++14"};
// access by const reference, s: const std::string&
for (const auto& s : vs)
  std::cout << s << '\n';

// range-for over initializer list
for (int n: {0, 1, 2, 3, 4, 5})
  std::cout << n << '\n';

// range-for over built-in array
int a[] = {0, 1, 2, 3, 4, 5}, sum{0};
for (int n: a)
  sum += n;
```

Don't Use Raw Loops

raw loop: any loop inside a function where the function serves a larger purpose than the algorithm implemented by the loop

- difficult to reason about and prove post-conditions
- error-prone and likely to fail under non-obvious conditions
- introduce non-obvious performance problems
- complicates reasoning about the surrounding code

Don't Use Raw Loops

raw loop: any loop inside a function where the function serves a larger purpose than the algorithm implemented by the loop

- difficult to reason about and prove post-conditions
- error-prone and likely to fail under non-obvious conditions
- introduce non-obvious performance problems
- complicates reasoning about the surrounding code

```
void func(std::vector<std::string>& v) {
    // ...
    std::string val;
    std::cin >> val;
    int index = 0;              // bad
    for (int i = 0; i < v.size(); ++i)
        if (v[i] == val) {
            index = i;
            break;
        }
    // ...
}
```

Alternatives to Raw Loops

- use an existing algorithm
- prefer standard algorithms if available
- implement a known algorithm as a general function

Alternatives to Raw Loops

- use an existing algorithm
- prefer standard algorithms if available
- implement a known algorithm as a general function

```
void func(std::vector<std::string>& v)
{
    // ...
    std::string val;
    std::cin >> val;
    auto p = std::find(v.cbegin(), v.cend(), val);  // better
    // ...
}
```

Alternatives to Raw Loops

- use an existing algorithm
- prefer standard algorithms if available
- implement a known algorithm as a general function

```
void func(std::vector<std::string>& v)
{
    // ...
    std::string val;
    std::cin >> val;
    auto p = std::find(v.cbegin(), v.cend(), val);  // better
    // ...
}
```

```
std::vector<int> nums = {1, 2, 3, 4 ,5, 6, 7, 8};
// increase all elements
std::for_each(nums.begin(), nums.end(), [](int& n){ n++; });
// find first even element
auto it = std::find_if(nums.begin(), nums.end(), [](int n){ return n % 2 == 0; });
```

Hands-on Example

stl.cpp: STL & algorithms

```
# clone (if you haven't already)
git clone git@github.com:yourname/cppworkshop.git
git clone https://github.com/apmanol/cppworkshop.git
cd cppworkshop

# build
make distclean
make

# run
./bin/stl
```

Initializer Lists and Uniform Initialization

C++98

```
int a[] = {1, 2, 3, 4, 5};
std::vector<int> v;
for (int i = 1; i <= 5; ++i)
   v.push_back(i);

std::map<int, std::string> labels;
labels.insert(make_pair(1, "Open"));
labels.insert(make_pair(2, "Close"));
labels.insert(make_pair(3, "Reboot"));

Vector normalize(const Vector& v)
{
  float inv_len = 1.f / length(v);
  return Vector(v.x*inv_len, v.y*inv_len,
      v.z*inv_len);
}

Vector x = normalize(Vector(2, 5, 9));
Vector y(4, 2, 1);
```

C++11

```
int a[] = {1, 2, 3, 4, 5};
std::vector<int> v = {1, 2, 3, 4, 5};



std::map<int, std::string> labels = {
  {1, "Open" },
  {2, "Close" },
  {3, "Reboot" }};

Vector normalize(const Vector& v)
{
  float inv_len {1.f / length(v)};
  return {v.x*inv_len, v.y*inv_len,
    v.z*inv_len};
}

Vector x = normalize({2, 5, 9});
Vector y{4, 2, 1};
```

std::initializer_list

std::initializer_list

An object of type *std::initializer_list<T>* is a lightweight proxy object that provides access to an array of objects of type const T.

```
template <class T>
struct S {
  S(std::initializer_list<T> l) : v_(l) {
    std::cout << "constructed with a " << l.size() << "- element list\n";
  }
  void append(std::initializer_list<T> l) {
    v_.insert(v_.end(), l.begin(), l.end());
  }
 private:
  std::vector<T> v_;
};

S<int> s = {1, 2, 3, 4, 5}; // copy list-initialization
s.append({6, 7, 8});        // list-initialization in function call
```

Uniform Initialization

- avoid narrowing errors when converting between integer types
- avoid narrowing errors when converting between floating-point types
- a floating-point value cannot be converted to an integer type
- an integer value cannot be converted to a floating-point type

```
int ival = 257;
double dval = 3.14;
char c = ival;  // narrowing: c becomes 1
int i = dval;   // truncation: i becomes 3
char c2 {ival}; // error: narrowing
int i2 {dval};  // error: truncation
std::vector<int> v = {1, 4.3, 4, 6.0}; // error: no double to int coversion

// don't mix std::initializer_list with auto
int n;
auto x = n;   // x: int
auto z = {n}; // z: std::initializer_list<int>

// watch greedy matching with ctors taking std::initializer_list's, e.g.,
std::vector<int> v1(10, 20); // 10 elements, each with value 20
std::vector<int> v2{10, 20}; // 2 elements, with values 10 & 20
```

Raw String Literals

- R"(ccc)" notation for a sequence of characters ccc
- no need to escape backslashes & quotes
- may contain newlines
- to match)" in ccc you may use R"*(ccc)*"

C++98

```
string t = "C:\\A\\B\\C\\D\\file1.txt";

string t2 = "1st line.\n2nd line.\n";



string s = "\\w\\\\\\w";
string s2 = "\"few\" \"quotes\" \"here\"";
```

C++11

```
string t = R"(C:\A\B\C\D\file1.txt)";

string t2 = R"(1st line.\n2nd line.\n)";

string t3 = R"(1st line.
2nd line.)";

string s = R"(\w\\\w)";
string s2 = R"("few" "quotes" "here")";
```

Variadic Templates

A template parameter pack is a template parameter that accepts zero or more template arguments (non-types, types, or templates). A template with at least one parameter pack is called a variadic template.

```
void print_impl() {
  std::cout << std::endl;
}

template <typename T, typename... Args>
void print_impl(const T& firstArg, Args... args) {
  std::cout << firstArg << ' ';
  print_impl(args...); // recursive call
}

template <typename... Args>
void print(Args... args) {
  std::cout << sizeof...(args) << ": ";
  print_impl(args...);
}

print("Hello", 3.14, 42); // output: 3: Hello 3.14 42
```

Alias Templates

- name that refers to a family of types
- not possible to partially or explicitly specialize an alias template

```
template <class T> struct MyAlloc {};
template <class T> using Vec = std::vector<T, MyAlloc<T>>;
Vec<int> v; // v: std::vector<int, MyAlloc<int>>

template <class T> using ptr = T*;
ptr<int> x; // x: int*

// member type alias, useful for generic programming
template <typename T>
struct Container {
    using value_type = T;
};

template <typename Cont>
void func(const Cont& c)
{
    typename Cont::value_type n;
}
```

Defaulted Functions

- C++ has six kinds of special member functions:
  - default constructors
  - destructors
  - copy constructors
  - copy assignment operators
  - move constructors
  - move assignment operators
- to declare a defaulted function, append the `=default;` specifier to the end of a function declaration, and the compiler will generate the default implementation

Deleted Functions

If, instead of a function body, the special syntax `=delete;` is used, the function is defined as deleted. Any use of a deleted function is ill-formed (the program will not compile).

```
class A {
  A(const A&) = delete;              // disallow copying
  A& operator=(const A&) = delete;   // disallow copying
};

struct B {
  B(float);          // can initialize with a float...
  B(long) = delete;  // ...but not with long
};

struct C {
  virtual ~C() = default;
};

struct D {
  void* operator new(std::size_t) = delete;
  void* operator new[](std::size_t) = delete;
};
D* p = new D; // error, attempts to call deleted D::operator new
```

In-class Initializers

C++98

```
class A {
 public:
  A() : a_(4), b_(2),
    h_("text1"), s_("text2") {}
  A(int a) : a_(a), b_(2),
    h_("text1"), s_("text2") {}
  A(const std::string& h) : a_(4), b_(2),
    h_(h), s_("text2") {}

 private:
  int a_;
  int b_;
  std::string h_;
  std::string s_;
};
```

C++11

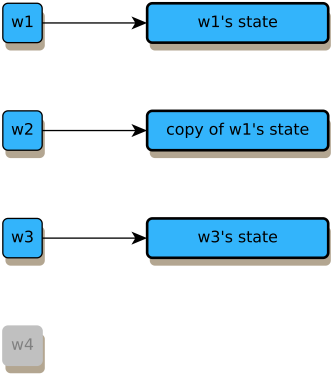
```
class A {
 public:
  A() {}

  A(int a) : a_{a} {}

  A(const std::string& h) : h_{h} {}


 private:
  int a_{4};
  int b_{2};
  std::string h_{"text1"};
  std::string s_{"text2"};
};
```

Move Semantics

- C++11 allows defining *move constructors* and *move assignments*, that move (rather than copy) their arguments
- move-enabled function overloads take their arguments by *non-const rvalue reference*; they can, and usually do, write to them
- move advantages:
  - cheap moving of a resource (instead of expensive copying)
  - non-copyable but moveable objects can be passed to (or returned from) a function by value
  - most standard types in C++11 are move-enabled

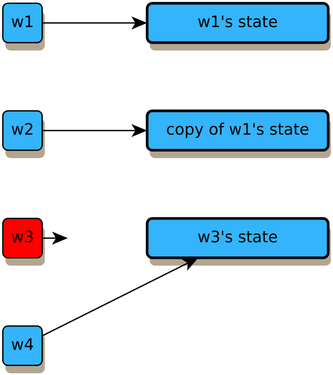
Move Semantics

```
Widget w1;
// ...

// copy w1's state to w2
Widget w2(w1);

Widget w3;
// ...

// move w3's state to w4
// Widget w4
```



Move Semantics

```
Widget w1;
// ...

// copy w1's state to w2
Widget w2(w1);

Widget w3;
// ...

// move w3's state to w4
Widget w4(std::move(w3));
```



Move Semantics

move from implementor's point of view

```
template <class T>
class vector
{
  // ...
  vector(const vector&);              // copy constructor
  vector(vector&&);                   // move constructor
  vector& operator=(const vector&); // copy assignment
  vector& operator=(vector&&);        // move assignment
  // ...
};
```

move from user's point of view

```
std::vector<std::string> colors;
std::string color{"red"};
colors.push_back(color);                 // calls push_back(const string&)
colors.push_back(std::move(color));      // calls push_back(string&&)
                                         // explicit, due to move()
// color will now be in a valid, but unspecified, state!
colors.push_back(std::string{"blue"}); // calls std::back(string&&)
                                         // implicit, due to temporary object
colors.emplace_back("green");            // object in-place constructed
```

Hands-on Example

move.cpp: filesystem access with move semantics

```
# clone (if you haven't already)
git clone git@github.com:yourname/cppworkshop.git
git clone https://github.com/apmanol/cppworkshop.git
cd cppworkshop

# build
make distclean
make

# build (enable AddressSanitizer)
make distclean
make debug=1

# run
./bin/move ./src/move.cpp
./bin/move /dev/urandom
```

Special Member Functions I

Special Member Functions

Constructors, Destructors, Assignment Operators

- compiler will generate special member functions, if they are needed and are not declared
- generated special member functions are implicitly `public` and `inline`, and they're *nonvirtual* unless for a destructor in a derived class inheriting from a base class with a virtual destructor
- generated **move operations** perform memberwise move requests on non-static data members via `std::move()`; for non-move-enabled types, these decay into copy operations

Special Member Functions II

- **move operations** are generated only for classes lacking explicitly declared move operations, copy operations, and a destructor
- **copy constructor** is generated only for classes lacking an explicitly declared copy constructor, and it's deleted if a move operation is declared
- **copy assignment operator** is generated only for classes lacking an explicitly declared copy assignment operator, and it's deleted if a move operation is declared
- generation of the **copy operations** in classes with an explicitly declared destructor or other copy operations is deprecated (hint: use default if you need them!)

Special Member Functions III

```
class Widget {
 public:
  ...
  ~Widget(); // user-declared dtor
  ...
  Widget(const Widget&) = default;
  Widget& operator=(const Widget&) = default;
};
```

- member function templates never suppress generation of special member functions

```
class Widget {
  ...
  template <typename T> Widget(const T& rhs);
  template <typename T> Widget& operator=(const T& rhs);
  // copy & move operations may still be generated!
};
```

- in C++11, generated destructors are `noexcept`

override

override specifier

Specifies that a virtual function overrides another virtual function.

C++98

```
struct Base {
  virtual bool prepare(int&);
  virtual bool commit(int);
};

struct Derived : Base {
  // compiles but doesn't run
  virtual bool prepare(int);
  // compiles but doesn't run
  virtual bool commit(int*);
};

Base* p = new Derived;
int id = 0;
if (p->prepare(id)) p->commit(id);
```

C++11

```
struct Base {
  virtual bool prepare(int&);
  virtual bool commit(int);
};

struct Derived : Base {
  // doesn't compile
  bool prepare(int) override;
  // doesn't compile
  bool commit(int*) override;
};

Base* p = new Derived;
int id = 0;
if (p->prepare(id)) p->commit(id);
```

final

final specifier

Specifies that a virtual function cannot be overridden in a derived class or that a class cannot be inherited from.

```
struct A
{
  virtual void foo() final;
  void bar() final; // ERROR: non-virtual function cannot be final
};

struct B final : A
{
  void foo(); // ERROR: A::foo() is final and cannot be overridden
};

struct C : B // ERROR: B is final and cannot be inherited
{
};
```

Scoped Enums

Scoped Enums

An *enumeration* is a distinct type whose value is restricted to one of several explicitly named constants (*enumerators*). The values of the constants are values of an integral type known as *the underlying type* of the enumeration.

```
enum class Alert { green, yellow, red };
enum class Color : int { red, blue };
int a1 = red;            // ERROR: red not in scope
int a2 = Alert::red;     // ERROR: no Alert -> int conversion
Color a3 = 7;            // ERROR: no int -> Color conversion
Color a4 = red;          // ERROR: red not in scope
Color a5 = Alert::red;   // ERROR: no Alert -> Color conversion
Color a6 = Color::blue;  // ok
```

constexpr

- constexpr *objects* are const and are initialized with values known during compilation
- constexpr *functions* can produce compile-time results when called with arguments whose values are known during compilation
- constructors can be constexpr, resulting to constexpr objects
- constexpr objects and functions may be used in a wider range of contexts than non-constexpr objects and functions (e.g., array sizes, integral template arguments, enumerator values etc)
- in C++11, constexpr functions are implicitly const; in C++14, they are not

constexpr

"If there were an award for the most confusing new word in C++11, constexpr would probably win it."

```
constexpr int x1 = std::ios_base::badbit | std::ios_base::eofbit;

void func(int flag) {
  constexpr int x2 = std::ios_base::badbit | flag;  // ERROR
  int x3 = std::ios_base::badbit | flag;
}

struct Point {
  int x, y;
  constexpr Point(int xx, int yy) : x{xx}, y{yy} {}
};

constexpr Point origo(0, 0);
constexpr int z = origo.x;

constexpr Point a[] = {Point(0, 0), Point(1, 1), Point(2, 2)};
constexpr int x = a[1].x;
```

nullptr

nullptr

- denotes the null pointer literal
- implicit conversions from nullptr to null pointer value of any pointer type and any pointer-to-member type

```
template <class F, class A>
void fwd(F f, A a) {
  f(a);
}

void g(int* i) {}

int main() {
  g(NULL);              // ok
  g(0);                 // ok
  g(nullptr);           // ok
  fwd(g, NULL);         // ERROR: no function g(long int)
  fwd(g, 0);            // ERROR: no function g(int)
  fwd(g, nullptr);      // ok
}
```

static_assert

static_assert

- performs compile-time assertion checking
- may appear at namespace and block scope (as a block declaration) and inside a class body (as a member declaration)

```
template <class T>
void f(T v)
{
  static_assert(sizeof(v) == 4, "v  must have size of 4 bytes");
  // ...
}

void g()
{
  int64_t v;  // v: 8 bytes
  f(v);       // compile error
}
```

Delegating Constructors

C++98

```
class A {
  int a_;
  void validate(int x) {
    if (0 < x && x <= 42)
      a_ = x;
    else
      throw bad_A(x);
  }

 public:
  A(int x) { validate(x); }
  A() { validate(42); }
  A(const std::string& s) {
    int x = std::stoi(s);
    validate(x);
  }
};
```

C++11

```
class A {
  int a_;

 public:
  A(int x) {
    if (0 < x && x <= 42)
      a_ = x;
    else
      throw bad_A(x);
  }
  A()
    : A(42) {}
  A(const std::string& s)
    : A(std::stoi(s)) {}
};
```

- cannot both delegate and explicitly initialize a member
- the object is not considered constructed until the delegating constructor completes

Inheriting Constructors

- equivalent to a constructor that simply initializes the base
- if a class adds data members to a base or requires a stricter class invariant, we should not inherit constructors

example

```
template <class T>
struct Vector : std::vector<T>
{
  using vector<T>::vector;
  // ...
};
```

Outline

Smart Pointers

- smart pointers are wrappers around raw pointers that act much like the raw pointers they wrap, but avoid many of their pitfalls
- there are three smart pointers classes:
  - `std::unique_ptr`
  - `std::shared_ptr`
  - `std::weak_ptr`
- advice: avoid calling `new` and `delete` explicitly

std::unique_ptr

- smart pointer that retains sole ownership of an object
- pointee is destroyed when the pointer goes out of scope
- no two `unique_ptr` instances can manage the same object

```
struct A { void foo() {} };

void bar(const A&) {}

int main() {
  std::unique_ptr<A> p1(new A);
  if (p1)
    p1->foo();
  {
    std::unique_ptr<A> p2(std::move(p1)); // transfer ownership
    bar(*p2);
    p1 = std::move(p2);  // ownership returns to p1
    std::unique_ptr<A> p3 = std::make_unique<A>();
    p3->foo();
  } // p3 is destroyed, and so is the A instance it owns
    // p2 is destroyed (owns nothing)
} // p1 is destroyed, and so is the A instance it owns
```

std::unique_ptr

- small, fast, move-only smart pointer for exclusive-ownership semantics
- allows custom deleter, whose type is specified in the std::unique_ptr template type parameters:

```
auto mydel = [](T* p) { log(*p); delete p; };
std::unique_ptr<T, decltype(mydel)> p{new T, mydel};
```

- std::unique_ptr can be converted to std::shared_ptr, so return them in factory functions:

```
std::unique_ptr<T> makeT() { /* make T factory */ }
std::unique_ptr<T> upt = makeT();
std::shared_ptr<T> spt = makeT();
```

- prefer std::make_unique() for more compact & exception safe code:

```
auto val{std::unique_ptr<T>{new T{}}};
auto val{std::make_unique<T>()}; // requires C++14!
```

std::shared_ptr

- smart pointer that retains shared ownership of an object
- several `shared_ptr` pointers may own the same object
- pointee is destroyed when either the last owner `shared_ptr` is destroyed, or the last remaining `shared_ptr` owning the object is assigned another pointer via `operator=` or `reset()`

```
void test() {
  std::shared_ptr<int> p(new int(42));
  assert(p.use_count() == 1);
  {
    std::shared_ptr<int> p2 = p;
    assert(p.use_count() == 2);
    {
      std::shared_ptr<int> p3 = p;
      assert(p.use_count() == 3);
    }
    assert(p.use_count() == 2);
  }
  assert(p.use_count() == 1);
} // p.use_count() == 0, delete p is called
```

std::shared_ptr

- compared to std::unique_ptr, std::shared_ptr objects are typically twice as big, incur overhead for control blocks, and require atomic reference count manipulations
- allows custom deleter, whose type has no effect on the type of the std::shared_ptr:

```
std::shared_ptr<T> p{new T, [](T* p) { log(*p); delete p; }};
```

- std::shared_ptr cannot be converted to std::unique_ptr
- prefer std::make_shared() for more compact & exception safe code:

```
tx(std::shared_ptr<Msg>{new Msg{}}, seq()); // may leak if seq() throws
tx(std::make_shared<Msg>(), seq()); // bonus: optimize control block allocation
```

Use RAII Pattern

RAII

“Resource Acquisition Is Initialization” is a basic technique for resource management based on scopes

explicit resource management

```
void f(const char* name)
{
  FILE* input = fopen(name, "r");
  // ...
  if (some_error())
    return; // bad: leaks file handle

  // bad: if an exception is thrown,
  // a file handle is leaked
  // ...

  fclose(input);
}
```

RAII

```
void f(const char* name)
{
  std::ifstream input{name};
  // ...
  if (some_error())
    return; // ok: no leak

  // ok: if an exception is thrown,
  // no file handle is leaked
  // ...

  // close file when ifstream destroyed
}
```

RAII Example

explicit resource management

```
void send(Message* msg,
          const std::string& dest) {
  static std::mutex mtx;
  PortHandle port = OpenPort(dest);
  mtx.lock();
  // ...
  Send(port, msg);
  // ...
  mtx.unlock();
  ClosePort(port);
  delete msg;
}
```

RAII Example

explicit resource management

```
void send(Message* msg,
          const std::string& dest) {
  static std::mutex mtx;
  PortHandle port = OpenPort(dest);
  mtx.lock();
  // ...
  Send(port, msg);
  // ...
  mtx.unlock();
  ClosePort(port);
  delete msg;
}
```

RAII

```
class Port {
  PortHandle port;
 public:
  Port(const std::string& dest)
    : port{OpenPort(dest)} {}
  ~Port() { ClosePort(port); }
  operator PortHandle() { return port; }
  Port(const Port&) = delete;
  Port& operator=(const Port&) = delete;
};

void send(std::unique_ptr<Message> msg,
          const std::string& dest) {
  static std::mutex mtx;
  Port port{dest};
  std::lock_guard<std::mutex> guard{mtx};
  // ...
  Send(port, msg.get());
  // ...
}
```

RAII Example

explicit resource management

```
void f(int i)
{
    int* p = new int[42];
    // ...
    if (i > 42) {
        delete p;
        throw Bad{"in f()", i};
    }
    // ...
}
```

RAII

```
void f(int i)
{
    auto p = std::make_unique<int[42]>();
    // ...
    if (i > 42) {
        throw Bad{"in f()", i};
    }
    // ...

}
```

RAII Summary

- any time you want to perform some action along every path out of a block, the normal approach is to put that action in the destructor of a local object
- STL examples applying RAII:
    - STL containers (destroy contents and release memory)
    - smart pointers (delete pointee, decrement refcount)
    - std::stream (close file)
    - std::lock_guard (unlock mutex)

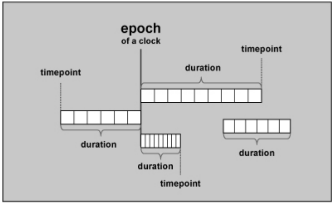
Hands-on Example

raii.cpp: network port scanning with RAII

```
# clone (if you haven't already)
git clone git@github.com:yourname/cppworkshop.git
git clone https://github.com/apmanol/cppworkshop.git
cd cppworkshop

# build
make distclean
make

# run
./bin/raii
valgrind ./bin/raii
```

std::chrono

- standard clock and timers
- *timepoint* defined as a *duration* before or after an epoch, which is defined by a *clock*
- *clock* defines an epoch and a tick period
- *duration* is a combination of a *value* representing the number of ticks and a *fraction* representing the unit in seconds
- `system_clock`: real-time clock
- `steady_clock`: monotonic clock



std::chrono

- ratios

```
using FiveThirds = std::ratio<5, 3>;
using AlsoFiveThirds = std::ratio<25, 15>;
std::ratio<42, 42> one;
std::cout << one.num << "/" << one.den << std::endl;
```

- durations

```
std::chrono::duration<int> twentySeconds{20};
std::chrono::duration<double, std::ratio<60>> halfAMinute{0.5};
std::chrono::duration<long, std::ratio<1, 1000>> oneMillisecond{1};
std::chrono::milliseconds ms{100};
ms += twentySeconds;
std::cout << ms.count() << " msec" << std::endl;
std::cout << std::chrono::nanoseconds(ms).count() << " nsec" << std::endl;
```

- clocks and timepoints

```
auto start = std::chrono::steady_clock::now();
// ...
auto elapsed = std::chrono::steady_clock::now() - start;
auto sec = std::chrono::duration_cast<std::chrono::seconds>(elapsed);
std::cout << "duration:" << sec.count() << " sec" << std::endl;
```

Iterators (Primer)

Iterators are

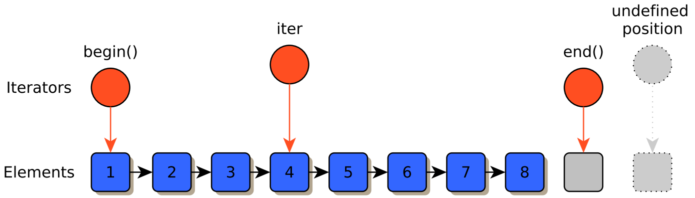
objects that can iterate over elements of a sequence. Iterators are the glue that holds containers and algorithms together. They provide an abstract view of the data. Anything that behaves like an iterator is an iterator.

- Most famous are `begin()` and `end()`.
- Basic properties are dereference `*itr` and advance `++itr`

Iterators (Primer)

Iterator Categories

- Input Iterator: Reads forward
- Output Iterator: Writes forward
- Forward Iterator: Reads and Writes forward
- Bidirectional Iterator: Reads and Writes forward and backward
- Random Access Iterator: Reads and Writes with random access



basic iterators

Algorithms

new C++11 algorithms

```
bool all_of(Iter first, Iter last, Pred pred);
bool any_of(Iter first, Iter last, Pred pred);
bool none_of(Iter first, Iter last, Pred pred);

Iter find_if_not(Iter first, Iter last, Pred pred);

OutIter copy_if(InIter first, InIter last, OutIter result, Pred pred);
OutIter copy_n(InIter first, InIter::difference_type n, OutIter result);

OutIter move(InIter first, InIter last, OutIter result);
OutIter move_backward(InIter first, InIter last, OutIter result);

pair<OutIter1, OutIter2> partition_copy(InIter first, InIter last,
                                        OutIter1 out_true, OutIter2 out_false,
                                        Pred pred);
Iter partition_point(Iter first, Iter last, Pred pred);

RAIter partial_sort_copy(InIter first, InIter last, RAIter result_first,
                         RAIter result_last);
RAIter partial_sort_copy(InIter first, InIter last, RAIter result_first,
                         RAIter result_last, Compare comp);
```

Algorithms

new C++11 algorithms (cont'd)

```
bool is_sorted(Iter first, Iter last);
bool is_sorted(Iter first, Iter last, Compare comp);
Iter is_sorted_until(Iter first, Iter last);
Iter is_sorted_until(Iter first, Iter last, Compare comp);

bool is_heap(Iter first, Iter last);
bool is_heap(Iter first, Iter last, Compare comp);
Iter is_heap_until(Iter first, Iter last);
Iter is_heap_until(Iter first, Iter last, Compare comp);

T min(initializer_list<T> t);
T min(initializer_list<T> t, Compare comp);
T max(initializer_list<T> t);
T max(initializer_list<T> t, Compare comp);
pair<const T&, const T&> minmax(const T& a, const T& b);
pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
pair<const T&, const T&> minmax(initializer_list<T> t);
pair<const T&, const T&> minmax(initializer_list<T> t, Compare comp);
pair<Iter, Iter> minmax_element(Iter first, Iter last);
pair<Iter, Iter> minmax_element(Iter first, Iter last, Compare comp);

void iota(Iter first, Iter last, T value);
```

Containers

std::array

`std::array` is a container that encapsulates fixed size arrays

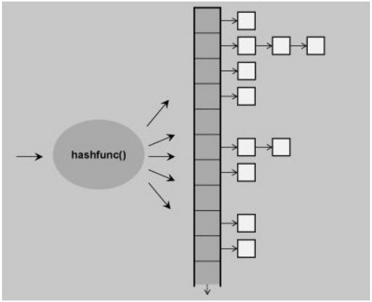
```
std::array<int, 3> a1{
    {1, 2, 3}};  // double-braces required in C++11 (not in C++14)
std::array<int, 3> a2 = {1, 2, 3};  // never required after =
std::array<std::string, 2> a3 = {std::string{"a"}, "b"};

// container operations are supported
std::sort(a1.begin(), a1.end());
std::reverse_copy(a2.begin(), a2.end(),
                  std::ostream_iterator<int>(std::cout, " "));
std::cout << '\n';

// ranged-for is supported
for (const auto& s : a3)
  std::cout << s << '\n';
```

Containers

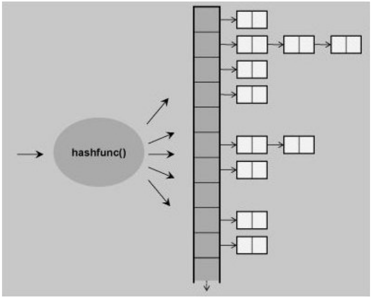
Unordered Containers

- an unordered container is a kind of a hash table
- C++11 offers four standard unordered containers:
  - `unordered_map`
  - `unordered_set`
  - `unordered_multimap`
  - `unordered_multiset`
- `unordered_map`
  - associative container that contains key-value pairs with unique keys
  - search, insertion, and removal of elements have average constant-time complexity (but worst linear!)
  - no sorting

Containers



unordered_set



unordered_map

Containers

	map	**unordered_map**
element ordering	strict weak	n/a
common implementation	balanced tree or red-black tree	hash table
search time	log(n)	O(1) if there are no hash collisions up to O(n) if there are hash collisions O(n) when hash is the same for any key
insertion time	log(n) + rebalance	same as search
deletion time	log(n) + rebalance	same as search
needs comparators	only operator <	only operator ==
needs hash function	no	yes
common use case	when good hash is not possible or too slow, or when order is required	in most other cases

map vs unordered_map

Concurrency

std::thread

- C++ handle to OS software thread
- `std::thread` *joinable* if connected to a software thread
- can be moved, but not copied
- use `std::ref()` to pass argument by reference

```
void func(int i) { std::cout << i << '\n'; }
struct functor { void operator()(int& i) { std::cout << ++i << '\n'; } };

std::vector<std::thread> threads;
std::thread thread{func, n};
threads.push_back(std::move(thread));
threads.push_back(std::thread{functor{}, std::ref(n)});
threads.push_back(std::thread{[](int i) { std::cout << i << '\n'; }, n + 2});
threads.emplace_back([](auto x) { std::cout << x << '\n'; }, n + 3);

for (auto& thread : threads)
  thread.join();
```

Concurrency

Synchronization

- `std::mutex`
  - thread synchronization facility
  - used to avoid data races when accessing shared data
  - API: `lock()/try_lock()/unlock()`
- `std::lock_guard` manages mutex lifetime according to RAII pattern (i.e., releases mutex upon scope exit)
- `std::unique_lock` like `std::lock_guard`, but allows more advanced use:
  - explicit set and release of the locks
  - moving and swapping locks
  - tentative or delayed locking

Concurrency

```
std::mutex m;

void func(size_t n) {
  std::lock_guard<std::mutex> lm{m}; // m.lock()
  for (size_t i = 0; i < 5; ++i) {
    std::cout << n << ": " << i << std::endl;
  }
} // m.unlock()

int main() {
  std::thread t1{func, 1};
  std::thread t2{func, 2};
  std::thread t3{func, 3};
  t1.join();
  t2.join();
  t3.join();
  return 0;
}
```

```
std::unique_lock<std::mutex> lk1{mutex1, std::defer_lock};
std::unique_lock<std::mutex> lk2{mutex2, std::defer_lock};
std::lock(lk1, lk2);
```

Hands-on Example

thread.cpp: apply multi-threading to a simple task

```
# clone (if you haven't already)
git clone git@github.com:yourname/cppworkshop.git
git clone https://github.com/apmanol/cppworkshop.git
cd cppworkshop

# build
make distclean
make

# run
./bin/thread
```

Outline

Guidelines Summary I

- no `define`: use constexpr, templates
- no naked `new`/`delete`: use smart pointers, RAII
- no C-style arrays: use STL containers (e.g., `std::array`)
- no ad-hoc, non-reusable functions: use (local) lambdas
- no type repetition: use `auto` wherever possible
- no NULL or 0 for pointers: use `nullptr`
- no raw loops: use STL algorithms, ranged-for
- no plain enums: use `enum class`
- no `typedef`: use `using` for type/template aliases
- performance: use move constructors/assignments to implement shallow (vs deep) copies

Guidelines Summary II

- inheritance: use `override` specifier when overriding virtual functions
- don't reinvent the wheel: use STL, boost
- use RAII pattern to avoid resource leaks
- declare variables as locally as possible & always initialize them
- use (preferably) `constexpr` or `const` whenever possible
- avoid hard-coded magic numbers
- avoid global variables/singletons and `extern`
- follow Single Responsibility Principle
- use clear and well defined interfaces
- prefer small and focused functions

Resources

Books

- Stroustrup: The C++ Programming Language (4th Ed)
- Josuttis: The C++ Standard Library (2nd Ed)
- Meyers: Effective Modern C++

Online Documentation

- cppreference.com
- Awesome C++
- Summary Of Changes
- Welcome Back To C++
- Manipulating C++ Containers Functionally
- C++ Core Guidelines
- Stroustrup C++11 FAQ