C++11
The Future is here

Bjarne Stroustrup
Texas A&M University
www.stroustrup.com
Overview

• What is C++?
• Making simple things simple
  – Uniform and universal initialization
  – Auto
  – Range-for
  – ...
• Resource Management
• Generic programming support
  – Lambdas
  – Variadic templates
  – Template aliases
  – ...
• Concurrency

Stroustrup - ACCU'13
What is C++?

- Template meta-programming!
- A hybrid language
- A multi-paradigm programming language
- It’s C!
- Embedded systems programming language
- Low level!
- A random collection of features

- Class hierarchies
- Generic programming
- Too big!
- Classes
- Buffer overflows

An object-oriented programming language

Stroustrup - ACCU'13
Programming Languages

- Assembler: Direct mapping to hardware
- Fortran: Domain-specific abstraction
- Cobol: Domain-specific abstraction
- Simula: General-purpose abstraction
- BCPL: General-purpose abstraction
- C: Direct mapping to hardware
- C++: General-purpose abstraction
- C++11: General-purpose abstraction
- Java: Domain-specific abstraction
- C#: Domain-specific abstraction

Stroustrup - ACCU'13
A light-weight abstraction programming language

Key strengths:
- software infrastructure
- resource-constrained applications

Stroustrup - ACCU'13
The ISO C++ Standard

- 1979 work on C with Classes starts
- 1985 first C++ commercial release
- 1990 work on an ANSI C++ standard starts
  - Based on “The ARM”
- 1998 first ISO C++ standard
- 2011 second ISO C++ standard
  - Compilers and libraries now available
- 2014 next ISO C++ revision

- No formal resources
  - No money, many volunteers
  - www.isocpp.org, The C++ Foundation
- 80 representatives present at meetings
  - 103+ in Bristol, April’13 – a new world record
- 250+ people involved
  - Much “electronic activity”
- Very democratic process
  - “herding cats”
Lists of C++11 features

• You know where to find them
  – E.g. www.stroustrup.com/C++11FAQ.html
  – GCC 4.7, Clang 3.1, ...

• What matter is how features work in combination
The real problems

• Help people to write better programs
  – Easier to write
  – Easier to maintain
  – Easier to achieve acceptable resource usage

• The primary value of a programming language is in the applications written in it

Stroustrup - ACCU'13
C++ applications
C++ Applications

- www.research.att.com/~bs/applications.html
C++ Applications

www.lextrait.com/vincent/implementations.html

Stroustrup - ACCU'13
C++11

- Is a better approximation of my ideals for support of good programming
  - Significantly better than C++98
- Has tons of distracting “old stuff”
  - Going back to C in 1972
- We must focus on the essentials
  - And the “good stuff”
  - “Elegance and efficiency”
- C++11 is not the end, we can do much better still
  - Anyone who says I have a perfect language is a fool or a salesman
- Stability/compatibility is an important feature in itself
  - And not free
Make simple tasks simple

- Uniform and universal initialization
- Auto
- Range-for
- User-defined literals
- Constexpr
Uniform initialization

- You can use `{}`-initialization for all types in all contexts
  
  ```
  int a[] = {1, 2, 3};
  vector<int> v { 1,2,3};
  
  vector<string> geek_heros = {
      "Dahl", "Kernighan", "Mcllroy", "Nygaard ", "Ritchie", "Stepanov"
  };
  
  thread t{}; // default initialization
  // remember “thread t();” is a function declaration
  
  complex<double> z{1,2}; // invokes constructor
  struct S { double x, y; } s {1,2}; // no constructor (just initialize members)
  ```
Uniform initialization

- `{}-initialization \( X\{v\} \) yields the same value of \( X \) in every context

\[
\begin{align*}
X & \ x\{a\}; \\
X^* & \ p = \text{new} \ X\{a\}; \\
z & = \ X\{a\}; & \quad & \text{\textit{// use as cast}} \\
\text{void} & \ f(X); \\
f(\{a\}); & \quad & \text{\textit{// function argument (of type X)}} \\
X & \ g() \{ \\
& \quad \text{\textit{// ...}} \\
& \quad \text{return} \ \{a\}; & \quad & \text{\textit{// function return value (function returning X)}} \\
& \} \\
Y::Y(a) : X\{a\} \{ /* \ldots */ \}; & \quad & \text{\textit{// base class initializer}}
\end{align*}
\]
auto

• Deduce a type of an object from its initializer
  
  auto x = 1;  // x is an int
  auto y = 1.2;  // y is a double

• Most useful when types get hard to type or hard to know

```cpp
template<class C>
void use(C& c)
{
  for (auto p = c.begin(); p!=c.end(); ++p)  // p is a ???
    cout << *p << ‘\n’;
}
```

• Curio: The oldest C++11 feature
  – I implemented it in 1983/84
range-for

• Make the simplest loops simpler

```cpp
template<class C>
void use(C& c)
{
    for (auto x : c)
        cout << x << ‘\n’;
}

for(auto x : { 1, 2, 5, 8, 13})
    test(x);
```

I ❤ C++
User-Defined Literals

• Examples
  – "Hello! " // const char*
  – "Howdy! "s // std::string
  – 2.3*5.7i // "i" for “imaginary”: a complex number
  – 4h+6min+3s // 4 hours, 6 minutes, and 3 seconds

• Can be used for type-rich programming
  – Speed s = 100m/9s; // very fast for a human
  – Acceleration a1 = s/9s; // OK
  – Acceleration a2 = s; // error: unit mismatch

• Definition
  – complex<double> operator "" i(long double d) { return {0,d}; }
General constant expressions

• Think
  – ROM
  – concurrency
  – Compile-time computation (performance, compactness)
  – Type safety (reliability, maintainability)

```cpp
constexpr int abs(int i) { return (0<=i) ? i : -i; } // can be constant expression

struct Point {
  int x, y;
  constexpr Point(int xx, int yy) : x{xx}, y{yy} { } // “literal type”
};

constexpr Point p1{1,2}; // must be evaluated at compile time: ok
constexpr Point p2{p1.y, abs(x)}; // ok?: is x is a constant expression?
```
Simplify Resource management and error handling

• Resources
  – A resource is something you acquire and must release
    • Release can (and should be implicit)
  – Never leak a resource

• RAII
  – Simplify code structure
  – Integrate resource management and error handling

• Move
  – Simplify interfaces
  – Don’t waste cycles

Stroustrup - ACCU'13
C++ Basics

- `int`, `double`, `complex<double>`, `Date`, ...
- `vector`, `string`, `thread`, `Matrix`, ...

Objects can be composed by simple concatenation:
- Arrays
- Classes/structs

If you understand `int` and `vector`, you understand C++
- The rest is “details” (1300 pages of details)
Resource management

• A resource should be owned by a “handle”
  – A “handle” should present a well-defined and useful abstraction
    • E.g. a vector, string, file, thread
• Use constructors and a destructor

```cpp
class Vector {
  // vector of doubles
  Vector(initializer_list<double>); // acquire memory; initialize elements
  ~Vector(); // destroy elements; release memory
  // ...

private:
  double* elem; // pointer to elements
  int sz; // number of elements
};

void fct()
{
  Vector v {1, 1.618, 3.14, 2.99e8}; // vector of doubles
  // ...
}
```

Value

handle
Resource management

• A resource should be owned by a “handle”
  – A “handle” should present a well-defined and useful abstraction
    • E.g. a vector, string, file, thread
• Use constructors and a destructor
  Vector::Vector(initializer_list<double> lst)
  : elem {new double[lst.size()], sz{lst.size()}; // acquire memory
  {
    uninitialized_copy(lst.begin(), lst.end(), elem); // initialize elements
  }
  
  Vector::~Vector()
  {
    delete[] elem; // destroy elements; release memory
  }
Resource management

• What about errors?
  – A resource is something you acquire and release
  – A resource should have an owner
  – Ultimately “root” a resource in a (scoped) handle
  – “Resource Acquisition is Initialization” (RAII)
    • Acquire during construction
    • Release in destructor
  – Throw exception in case of failure to construct (acquire)
  – Never throw while holding a resource not owned by a handle
Resource management

• For all resources
  – Memory (done by std::string, std::vector, std::map, ...)
  – Locks (e.g. std::unique_lock), files (e.g. std::fstream), sockets, threads (e.g. std::thread), ...

```cpp
std::mutex mtx; // a resource
int sh; // shared data

void f()
{
    std::lock_guard lck {mtx}; // grab (acquire) the mutex
    sh+=1; // manipulate shared data
} // implicitly release the mutex
```
Resource Handles and Pointers

• Many (most?) uses of pointers in local scope are not exception safe

```c++
void f(int n, int x)
{
    Gadget* p = new Gadget{n};  // look I’m a java programmer! 😊
    // …
    if (x<100) throw std::runtime_error{“Weird!”};  // leak
    if (x<200) return;  // leak
    // …
    delete p;  // and I want my garbage collector! 😞
}
```

– “Naked New”! (bad idea)
– But, why use a “naked” pointer?

Stroustrup - ACCU'13
Resource Handles and Pointers

- A `std::shared_ptr` releases its object at when the last `shared_ptr` to it is destroyed

```cpp
void f(int n, int x)
{
    shared_ptr<Gadget> p {new Gadget{n}}; // manage that pointer!
    // ...
    if (x<100) throw std::runtime_error{“Weird!”}; // no leak
    if (x<200) return; // no leak
    // ...
}
```

- `shared_ptr` provides a form of garbage collection
  - For good *and* bad
  - But I’m not sharing anything
    - use a `unique_ptr`
Resource Handles and Pointers

• But why use a pointer at all?
• If you can, just use a scoped variable

```cpp
void f(int n, int x)
{
    Gadget g {n};
    // ...
    if (x<100) throw std::runtime_error{“Weird!”};  // no leak
    if (x<200) return;  // no leak
    // ...
}
```
Why do we use pointers?

• And references, iterators, etc.

• To represent ownership
  – Don’t! use handles

• To reference resources
  – from within a handle

• To represent positions
  – Be careful

• To pass large amounts of data (into a function)
  – E.g. pass by `const` reference

• To return large amount of data (out of a function)
  – Don’t
How to move a resource

• Common problem:
  – How to get a lot of data cheaply out of a function

• Idea #1:
  – Return a pointer to a new’d object
    
    Matrix* operator+(const Matrix&, const Matrix&);
    Matrix& res = *(a+b); // ugly! (unacceptable)

• Who does the delete?
  – there is no good general answer
How to move a resource

• Common problem:
  – How to get a lot of data cheaply out of a function

• Idea #2
  – Return a reference to a new’d object

    Matrix& operator+(const Matrix&, const Matrix&);
    Matrix res = a+b; // looks right, but ...

• Who does the delete?
  – What delete? I don’t see any pointers.
  – there is no good general answer
How to move a resource

• Common problem:
  – How to get a lot of data cheaply out of a function

• Idea #3
  – Pass an reference to a result object

    ```
    void operator+(const Matrix&, const Matrix&, Matrix& result);
    Matrix res = a+b; // Oops, doesn’t work for operators
    Matrix res2;
    operator+(a,b,res2); // Ugly!
    ```

• We are regressing towards assembly code
How to move a resource

• Common problem:
  – How to get a lot of data cheaply out of a function

• Idea #4
  – Return a Matrix
    Matrix operator+(const Matrix&, const Matrix&);
    Matrix res = a+b;

  • Copy?
    – expensive

  • Use some pre-allocated “result stack” of Matrixes
    – A brittle hack

  • Move the Matrix out
    – don’t copy; “steal the representation”
    – Directly supported in C++11 through move constructors
Move semantics

• Return a **Matrix**

  ```
  Matrix operator+(const Matrix& a, const Matrix& b)
  {
    Matrix r;
    // copy a[i]+b[i] into r[i] for each i
    return r;
  }
  
  Matrix res = a+b;
  ```

• Define move a constructor for **Matrix**
  – don’t copy; “steal the representation”
Move semantics

• Direct support in C++11: Move constructor

```cpp
class Matrix {
    Representation rep;
    // ...
    Matrix(Matrix&& a) // move constructor
    {
        rep = a.rep;  // *this gets a’s elements
        a.rep = {};   // a becomes the empty Matrix
    }
};
```

Matrix res = a+b;

Stroustrup - ACCU'13
RAII and Move Semantics

• All the standard-library containers provide it
  • vector
  • list, forward_list (singly-linked list), ...
  • map, unordered_map (hash table), ...
  • set, multi_set, ...
  • ...
  • string

• So do other standard resources
  • thread, lock_guard, ...
  • istream, fstream, ...
  • unique_ptr, shared_ptr
  • ...

Stroustrup - ACCU'13
Better Support for Generic Programming

- Lambdas
- Variadic templates
- Template aliases
- Type traits
Lambda expressions

• A lambda expression ("a lambda") is a use-once function object

```cpp
template<class C, class Oper>
void for_all(C& c, Oper op) // assume that C is a container of pointers
{
    for (auto& x : c)
        op(*x); // pass op() a reference to each element pointed to
}

void user()
{
    vector<unique_ptr<Shape>> v;
    while (cin)
    {
        v.push_back(read_shape(cin)); // read shape from input
        for_all(v, [](Shape& s){ s.draw(); }); // draw_all()
        for_all(v, [](Shape& s){ s.rotate(45); }); // rotate_all(45)
    }
}
Variadic templates

- Any number of arguments of any types

```cpp
template <class F, class ...Args>  // thread constructor
    explicit thread(F&& f, Args&&... args);  // argument types must
                                                // match the operation’s
                                                // argument types
```

```cpp
void f0();  // no arguments
void f1(int);  // one int argument
```

```cpp
thread t1 {f0};  // error: too many arguments
thread t2 {f0,1};  // error: too few arguments
thread t3 {f1};
thread t4 {f1,1};  // error: too many arguments
thread t5 {f1,1,2};  // error: too many arguments
thread t3 {f1,"I'm being silly"};  // error: wrong type of argument
```
Template aliases

- Notation matters
- C++98 exposes all details when we use templates
  \[
  \text{typename iterator_traits<For>::value_type } x;
  \]
- C++11 allows us to hide details
  \[
  \text{template<typename Iter>}
  
  \text{using Value_type<T> } = \text{typename std::iterator_traits<For>::value_type;}
  
  \text{// \ldots}
  
  \text{Value_type<For> } x;
  \]
- Had I had an initializer, I could have used \text{auto}
  \[
  \text{auto } x = *p;
  \]
Range for and move

• As ever, what matters is how features work in combination

```cpp
template<typename C, typename V>
vector<Value_type<C>*>* find_all(C& c, V v) // find all occurrences of v in c
{
    vector<Value_type<C>*>* res;
    for (auto& x : c)
        if (x==v)
            res.push_back(&x);
    return res;
}
```

```cpp
string m {"Mary had a little lamb"};
for (const auto p : find_all(m,'a')) // p is a char*
    if (*p!='a')
        cerr << "string bug!\n";
```
Don’t start from the bare language

• Some standard-library components
  – Type-safe concurrency
    • Conventional threads and locks
    • Futures and async()
  – Regular expressions
  – Hash tables
    • Yes, they weren’t standard until C++11
  – Random numbers
  – STL
    • Many “small” improvements
      – New algorithms, containers, functions
      – Move semantics

Stroustrup - ACCU'13
Concurrency

- There are many kinds
- Stay high-level
- Stay type-rich
Type-Safe Concurrency

• Programming concurrent systems is hard
  – We need all the help we can get
  – C++11 offers
    • A memory model for concurrency
    • Support for lock-free programming
    • type-safe programming at the threads-and-locks level
    • One simple higher-level model (futures and async task launching)
  – Type safety is hugely important

• threads-and-locks
  – is an unfortunately low level of abstraction
  – is necessary for current systems programming
    • That’s what the operating systems offer
  – presents an abstraction of the hardware to the programmer
  – can be the basis of other concurrency abstractions
Threads

```cpp
void f(vector<double>&);  // function

struct F {  // function object
    vector<double>& v;
    F(vector<double>& vv) : v{vv} { }
    void operator()();
};

void code(vector<double>& vec1, vector<double>& vec2)
{
    std::thread t1 {f, vec1};  // run f(vec1) on a separate thread
    std::thread t2 {F{vec2}};  // run F{vec2}() on a separate thread
    t1.join();
    t2.join();
    // use vec1 and vec2
}
```
Thread – pass argument and result

double* f(const vector<double>& v); // read from v return result
double* g(const vector<double>& v); // read from v return result

void user(const vector<double>& some_vec) // note: const
{
    double res1, res2;
    thread t1 {[&]{ res1 = f(some_vec); }}; // lambda: leave result in res1
    thread t2 {[&]{ res2 = g(some_vec); }}; // lambda: leave result in res2
    // ...
    t1.join();
    t2.join();
    cout << res1 << ' ' << res2 << '
';
}
async() — pass argument and return result

double* f(const vector<double>& v);  // read from v return result
double* g(const vector<double>& v);  // read from v return result

void user(const vector<double>& some_vec)          // note: const
{
    auto res1 = async(f,some_vec);
    auto res2 = async(g,some_vec);
    // ...
    cout << *res1.get() << ' ' << *res2.get() << '\n';    // futures
}

• Much more elegant than the explicit thread version
  – And most often faster
When? – Now!

• The compilers are getting good
  – Much faster adoption than C++98

• Use will lag for years
  – Decades?
  – Developers are very busy and can be very conservative
  – Teaching materials (even “new” ones)
  – Courses
  – Tools

• Fight FUD!
  – Start with the “low-hanging fruit” to gain credibility
Questions?

• Stroustrup: “A Tour of C++”
  http://isocpp.org/tour