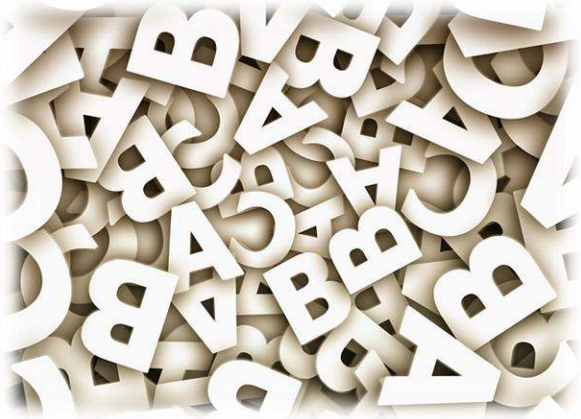


New C++ Style

„C++11 feels like a new language…”
– Bjarne Stroustrup

Mateusz Pusz
Intel Technology Poland

To the extent possible under law, Intel Corporation has waived all copyright and related or neighboring rights to this work.

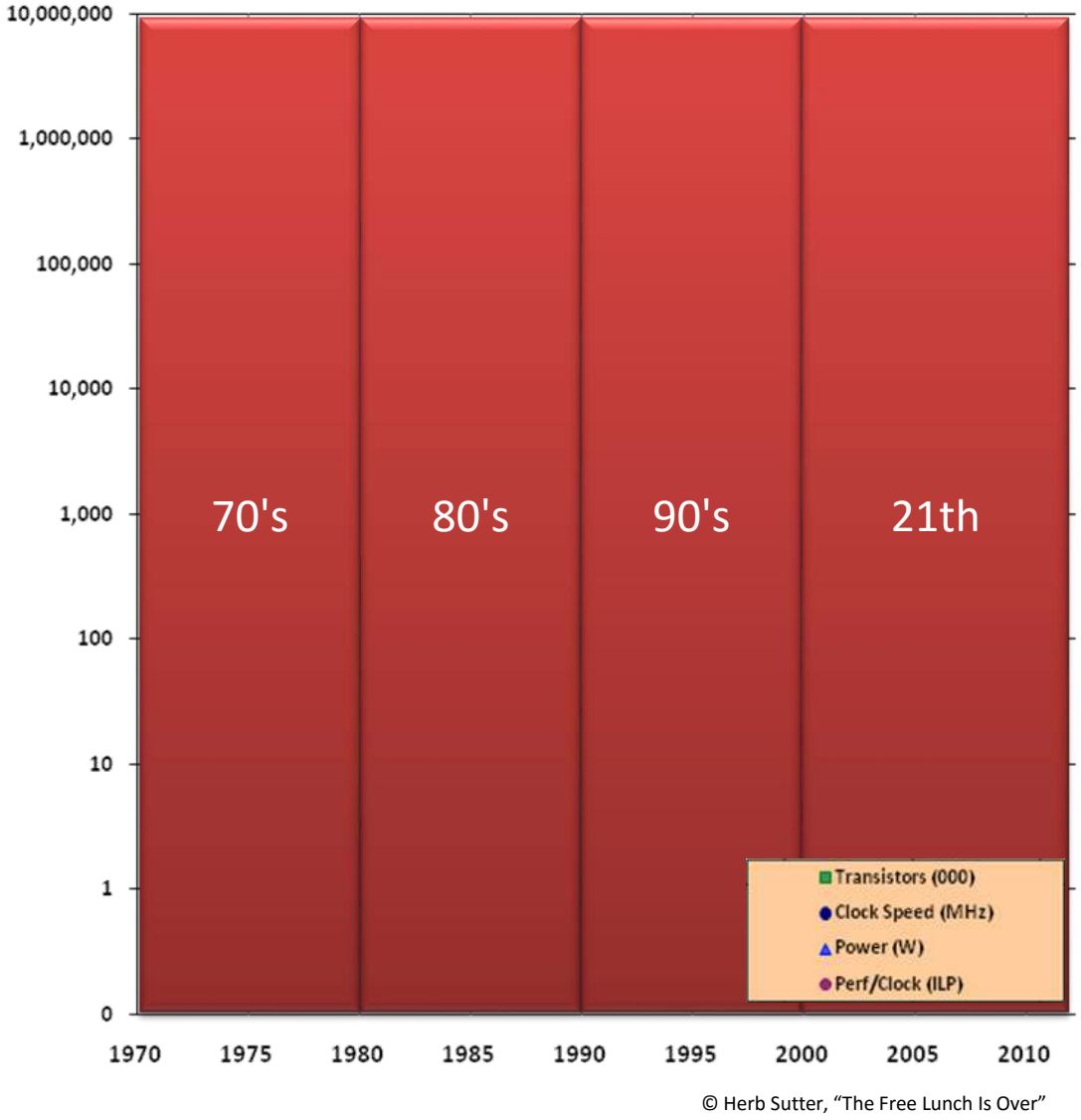
Some images used and marked with © are the subject of other licensing terms thus are not released to the public domain.

The Free Lunch is Over

"The free lunch is over.
Now welcome to the hardware jungle."

- Herb Sutter, December 2011

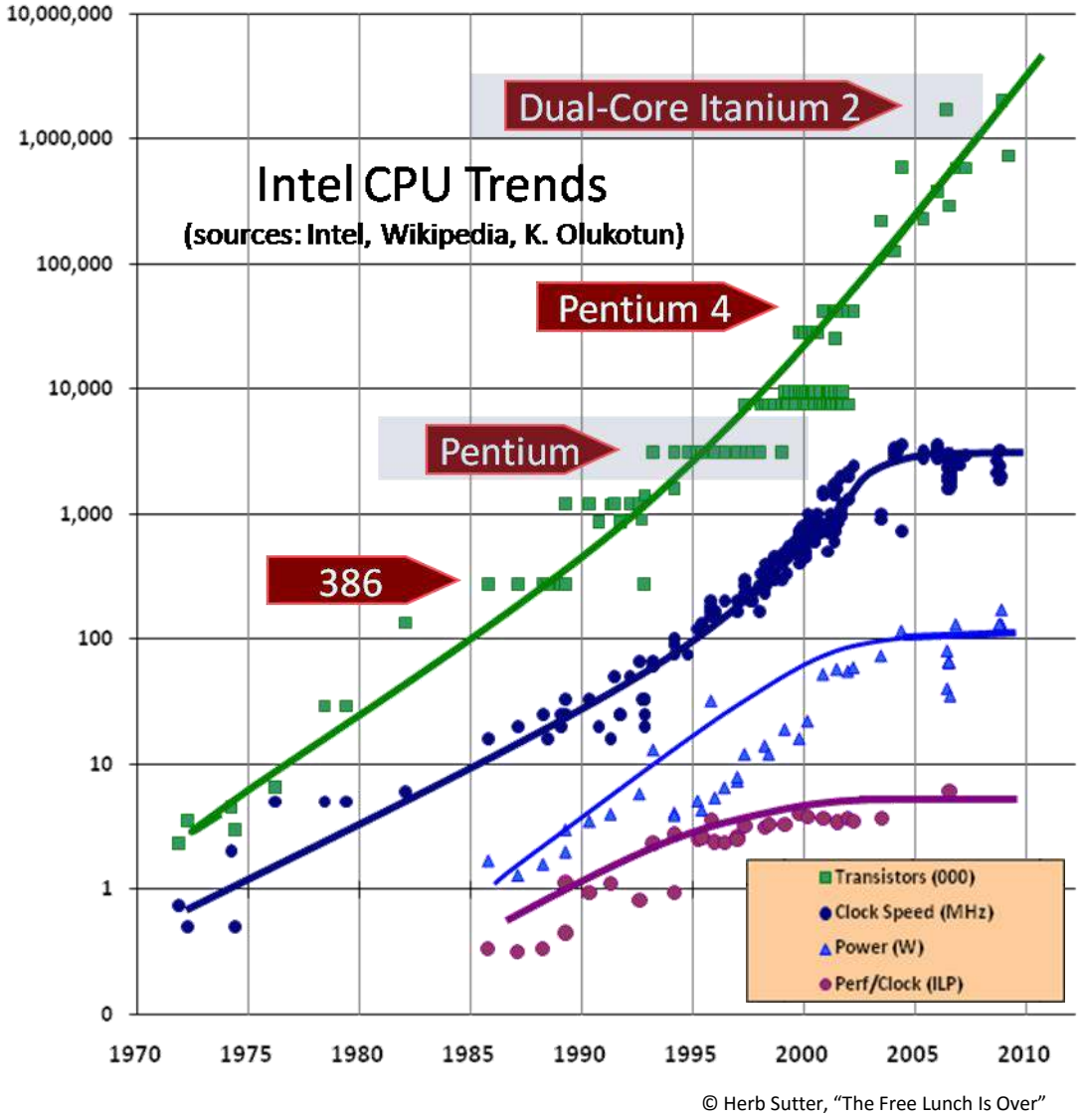
What was a Free Lunch?



© Herb Sutter, "The Free Lunch Is Over"

- **For long years clock speeds were increasing similarly to the number of transistors on a die (Moore's Law)**
  - once written applications were getting faster for free over time
  - high productivity "Coffee-based" languages golden era

What was a Free Lunch?

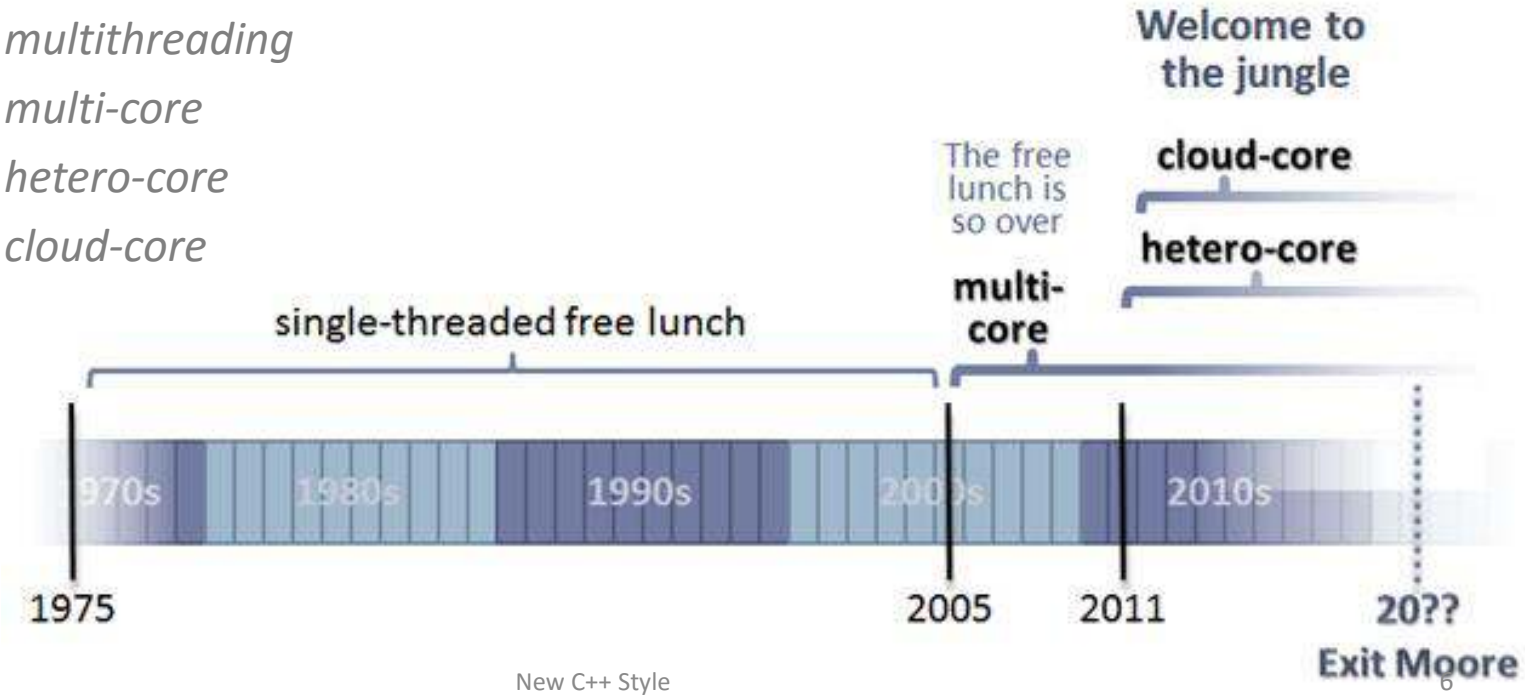


© Herb Sutter, "The Free Lunch Is Over"

- **For long years clock speeds were increasing similarly to the number of transistors on a die (Moore's Law)**
  - once written applications were getting faster for free over time
  - high productivity "Coffee-based" languages golden era

Where we are now?

- **In 2004 clock speeds suddenly stopped to grow**
- **To keep increase of performance we need to:**
  - invest in high-performance computing languages
    - *"Coffee-based" languages get less attention*
  - make high-performance languages more user-friendly
  - transition to parallel computing
    - *multithreading*
    - *multi-core*
    - *hetero-core*
    - *cloud-core*



© Herb Sutter, "Welcome to the Jungle"

Why C++?

mobile devices lifetime
big server farms maintenance cost
everyone's power bills

hardware costs
limited resources on mobiles

C

Fortran

get the most from each thread
bigger experiences on a smaller hardware

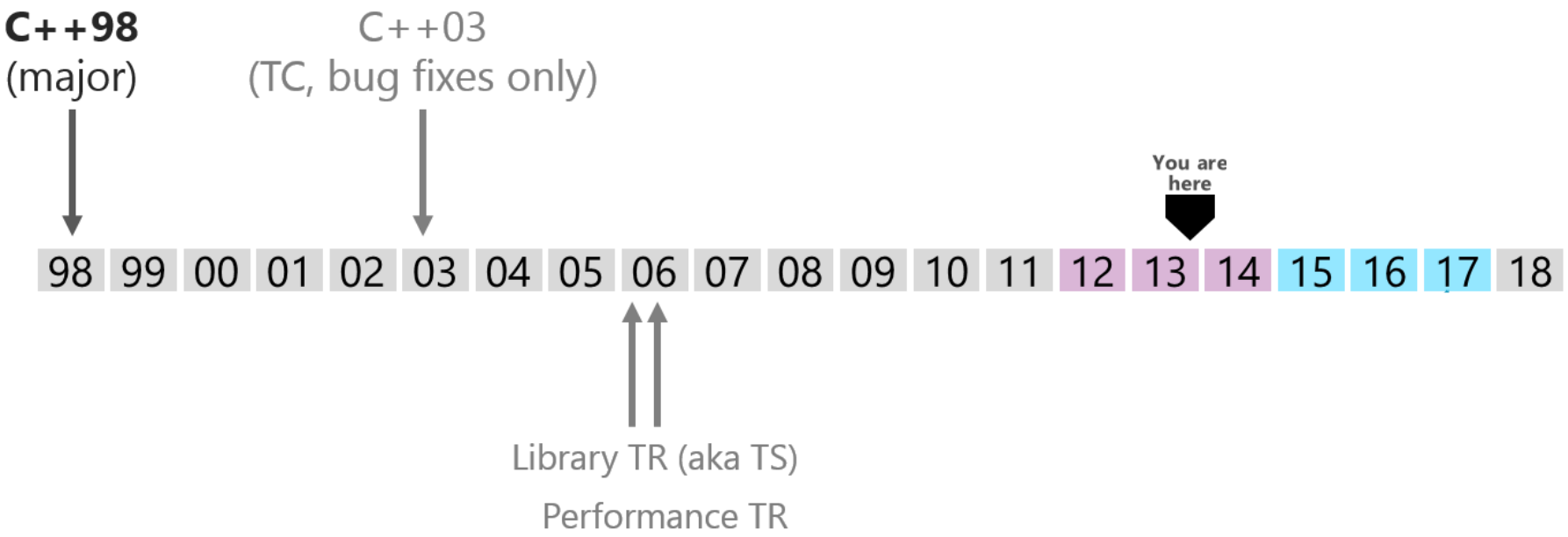
C++ Renaissance: the return of the King!

- **The decade where productivity was of the highest priority ends now**
- **Performance is the King, again!!!**
- **All major players switch to C++**
  - Microsoft again preferring C++ over C# (after Vista fiasco)
  - Mobiles (iPhone, Android) enable coding in native languages
  - Google, Facebook and Amazon have most of their code written in C++
- **A lot of money is invested now in C++**
  - it's cheaper to invest in developers and language features than in hardware and power (~88% costs of big data centers are performance and power related)
- **Scope on making C++**
  - even more efficient
  - more productive to be competitive to "Coffee-based" languages

C++11 (12 August 2011, ISO/IEC 14882:2011)

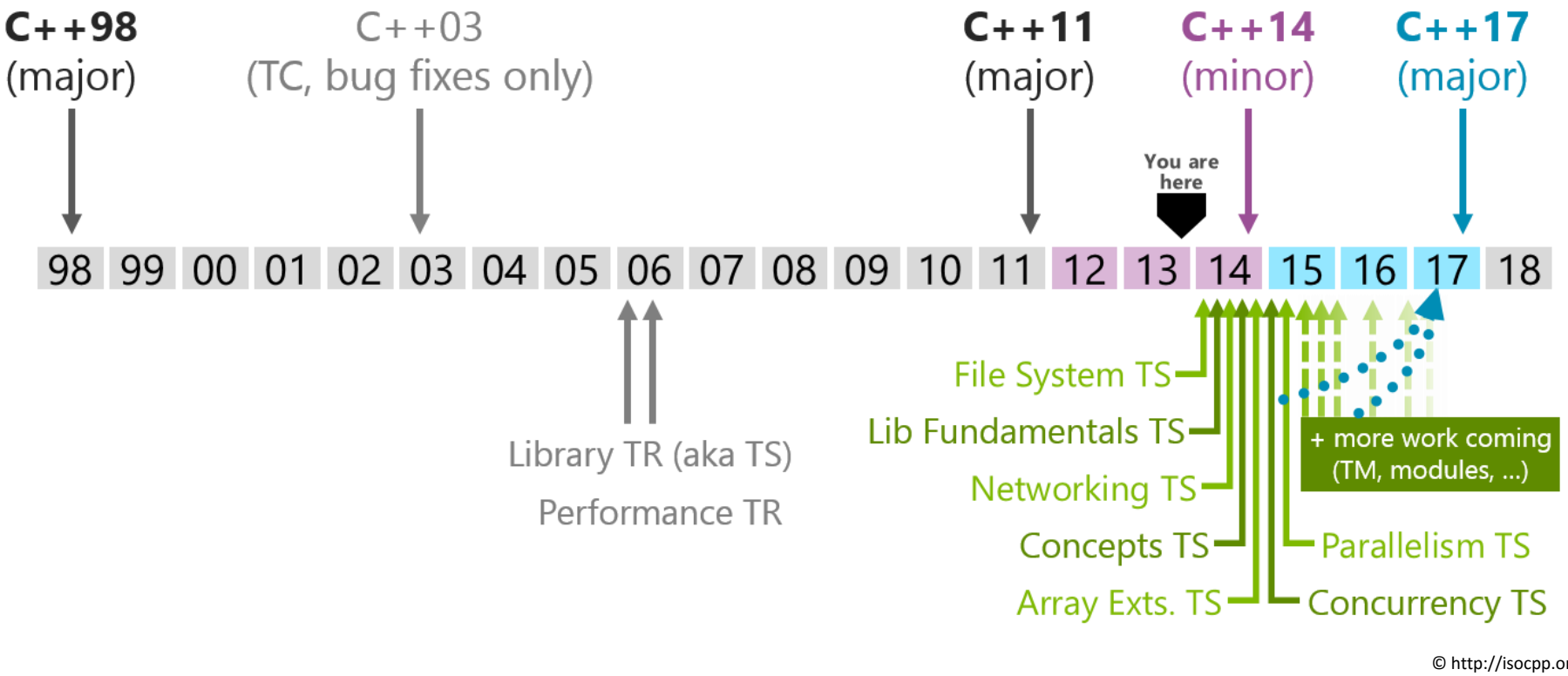
Core language runtime performance	• *Move Semantics* • *Constant Expressions*
Core language usability	• *Lambdas* • *Object construction improvements, initializer lists and explicit overrides* • *Range-based-for-loops* • *Strongly typed enumerations and null pointer constant*
Core language functionality	• *Variadic templates* • *Multitasking memory model* • *Explicitly defaulted and deleted special member functions* • *Static assertions*
C++ Standard Library	• *Threading facilities* • *More containers and new smart pointers* • *Type traits* • *Toolsets like: regular expressions, time utilities, random numbers*

C++ is far from being a dead language



© http://isocpp.org

C++ is far from being a dead language



© http://isocpp.org

C++11 vs Global Warming

„My contribution to the fight against global warming is C++'s efficiency.

Just think if Google had to have twice as many server farms! Each uses as much energy as small town. And it's not just factor of two...

Efficiency is not just running fast or running bigger programs, it's also running using less resources."

– Bjarne Stroustrup, June 2011

Readability vs Performance

Old

```
typedef std::vector<std::string> Strings;
void get_names(Strings &out_param);
void modify_copy(const Strings &names,  Strings
&out_param);
```

```
Strings names;
get_names(names);
Strings s1;
modify_copy(names, s1);
Strings s2;
{
   Strings temp_names;
   get_names(temp_names);
   modify_copy(temp_names, s2);
}
get_names(s1);      // who should clean the container?
```

New

```
typedef std::vector<std::string> Strings;
Strings get_names();
Strings modify_copy(Strings names);
```

```
const Strings names = get_names();
Strings s1 = modify_copy(names);
Strings s2 = modify_copy(get_names());
s1 = get_names();
```

Measured for 3 names	Old	New
Default-constructor	9	9
Copy-constructor	15	3
Destructor	24	12

- Modern C++ style makes the code much easier to understand making it much faster at the same time!!!

Readability vs Performance

Old

```
typedef std::vector<std::string> Strings;
void get_names(Strings &out_param);
void modify_copy(const Strings &names,  Strings
&out_param);
```

```
Strings names;
get_names(names);
Strings s1;
modify_copy(names, s1);
Strings s2;
{
   Strings temp_names;
   get_names(temp_names);
   modify_copy(temp_names, s2);
}
get_names(s1);      // who should clean the container?
```

New

```
using Strings = std::vector<std::string>;
Strings get_names();
Strings modify_copy(Strings names);
```

```
const Strings names = get_names();
auto s1 = modify_copy(names);
auto s2 = modify_copy(get_names());
s1 = get_names();
```

Measured for 3 names	Old	New
Default-constructor	9	9
Copy-constructor	15	3
Destructor	24	12

- Modern C++ style makes the code much easier to understand making it much faster at the same time!!!

Is it possible not to do anything at all?

Old

Runtime

```
bool is_prime(size_t number)
{
   if(number <= 1)
      return false;
   for(size_t  i=2; i*i <= number;  ++i)
      if(number % i == 0)
         return false;
   return true;
}
```

Compile-time

50 lines of hardcore template metaprogramming

Runtime

```
for(size_t i=0; i<10; i++)
   std::cout << is_prime(i) << '\n';
```

Compile-time

```
std::cout << is_prime_t<1>::value << '\n';
std::cout << is_prime_t<131>::value << '\n';
std::cout << is_prime_t<4256233>::value << '\n';
```

Is it possible not to do anything at all?

Old

Runtime

```
bool is_prime(size_t number)
{
   if(number <= 1)
      return false;
   for(size_t  i=2; i*i <= number;  ++i)
      if(number % i == 0)
         return false;
   return true;
}
```

Compile-time

50 lines of hardcore template metaprogramming

Runtime

```
for(size_t i=0; i<10; i++)
   std::cout << is_prime(i) << '\n';
```

New

Runtime & Compile-time

```
constexpr bool is_prime_impl(size_t n,  size_t c)
{
   return (c*c > n) ? true :
                (n % c == 0) ? false :
                   is_prime_impl(n,  c+1);
}
constexpr bool is_prime(size_t n)
{
   return (n <= 1) ? false : is_prime_impl(n, 2);
}
```

Compile-time

```
std::cout << is_prime(1) << std::endl;
std::cout << is_prime(131) << std::endl;
std::cout << is_prime(4256233) << std::endl;
```

- **New C++ does a lot of work in compile-time so the runtime is even faster than before!!!**

Is it possible not to do anything at all?

Old

Runtime

```
bool is_prime(size_t number)
{
   if(number <= 1)
      return false;
   for(size_t  i=2; i*i <= number;  ++i)
      if(number % i == 0)
         return false;
   return true;
}
```

Compile-time

50 lines of hardcore template metaprogramming

Future (C++14)

Runtime & Compile-time

```
constexpr bool is_prime(size_t number)
{
   if(number <= 1)
      return false;
   for(size_t  i=2; i*i <= number;  ++i)
      if(number % i == 0)
         return false;
   return true;
}
```

Runtime

```
for(size_t i=0; i<10; i++)
   std::cout << is_prime(i) << '\n';
```

Compile-time

```
std::cout << is_prime(1) << std::endl;
std::cout << is_prime(131) << std::endl;
std::cout << is_prime(4256233) << std::endl;
```

- C++14 makes it even more user friendly

Chasing "Coffee-based" Languages

„C++11 feels like a new language:
The pieces just fit together better than they used to and
I find a higher-level style of programming more natural than
before and as efficient as ever.”
– Bjarne Stroustrup

Writing shorter code

Old

```
int a[] = { 4, 2, 1, 3, 5 };
for(size_t i=0; i<sizeof(a)/sizeof(a[0]); ++i)
   a[i] *= 2;
sort(&a[0], &a[0] + sizeof(a)/sizeof(a[0]));
```

```
vector<int> v;
v.push_back(4);
v.push_back(2);
v.push_back(1);
v.push_back(3);
v.push_back(5);

for(vector<int>::iterator it = v.begin(); it!=v.end(); ++it)
   *it *= 2;

sort(v.begin(), v.end());
```

New

```
int a[] = { 4, 2, 1, 3, 5 };
for(auto &i : a)
   i *= 2;
sort(begin(a), end(a));
```

```
vector<int> v{ 4, 2, 1, 3, 5 };

for(auto &i : v)
   i *= 2;

sort(begin(v), end(v));
```

- **Shorter code doing the same is always better!!!**

Algorithms are finally fully usable

```
std::vector<int> get_data();

// find the first element in range (5, 10)
```

Old

```
struct my_cmp {
  int _x;  int _y;
  my_cmp(int x, int y): _x(x), _y(y) {}
  bool operator() (int i) const { return i > _x && i < _y; }
};
```

```
std::vector<int> v = get_data();
my_cmp cmp(5, 10);
std::vector<int>::iterator it = std::find_if(v.begin(), v.end(), cmp);
```

```
std::vector<int> v = get_data();
std::vector<int>::iterator it = v.begin();
for(; it!=v.end(); ++it) {
   if(*it > 5 && *it < 10)
      break;
}
```

New

```
auto v = get_data();
auto it = std::find_if(begin(v),  end(v),  [](int i) { return i > 5 && i < 10; } );
```

What? Where? How?

Algorithms are finally fully usable

```
std::vector<int> get_data();

// find the first element in range (5, 10)
```

Old

```
struct my_cmp {
  int _x;  int _y;
  my_cmp(int x, int y): _x(x), _y(y) {}
  bool operator() (int i) const { return i > _x && i < _y; }
};
```

```
std::vector<int> v = get_data();
my_cmp cmp(5, 10);
std::vector<int>::iterator it = std::find_if(v.begin(), v.end(), cmp);
```

```
std::vector<int> v = get_data();
std::vector<int>::iterator it = v.begin();
for(; it!=v.end(); ++it) {
   if(*it > 5 && *it < 10)
      break;
}
```

Future (C++14)

```
auto v = get_data();
auto it = std::find_if(begin(v),  end(v),  [](auto& i) { return i > 5 && i < 10; } );
```

What? Where? How?

Who needs garbage collectors?

Old

```
void store(int *i,  double *d);
```

~~store(new int(10), new double(3.14));~~

```
{
   int *i = new int(10);
   double *d = 0;
   try {
      d = new double(3.14);
   }
   catch(const std::bad_alloc &) {
      delete i;
      throw;
   }
   store(i, d);
}
```

New

```
void store(std::unique_ptr<int> i,
           std::unique_ptr<double> d);
```

```
store(make_unique<int>(10),
      make_unique<double>(3.14));
```

„C++ is the best language for garbage collection principally because it creates less garbage."

Bjarne Stroustrup

- Standardized C++ smart pointers take care about resource management for you and they are really good at it!

Who needs garbage collectors?

Old

```
void store(int *i,  double *d);
```

~~store(new int(10), new double(3.14));~~

```
{
   int *i = new int(10);
   double *d = 0;
   try {
      d = new double(3.14);
   }
   catch(const std::bad_alloc &) {
      delete i;
      throw;
   }
   store(i, d);
}
```

Future (C++14)

```
void store(std::unique_ptr<int> i,
           std::unique_ptr<double> d);
```

```
store(std::make_unique<int>(10),
      std::make_unique<double>(3.14));
```

„C++ is the best language for garbage collection principally because it creates less garbage.”

Bjarne Stroustrup

- In Modern C++ explicit use of 'new' and 'delete' operators should be avoided in most of the code!!!

Is memory the only resource to cleanup?

Old

```
void process_file(const char *path)
{
   FILE *f = 0;
   try {
      f = fopen(path, "r");
      // use f
   }
   catch(...) {  // handle every exception
      if(f)  fclose(f);
      throw;
   }
   if(f)  fclose(f);
}
```

New

```
using unique_file_ptr = std::unique_ptr<FILE, int (*)(FILE *)>;

void process_file(const char *path)
{
   unique_file_ptr  f(fopen(path, "r"),  fclose);
   // use f
}
```

- The same RAII-based approach can be seen anywhere in modern C++

Easy concurrency

Old

No standard way to handle multithreading before C++11!!!

New

```
// read from v and return result
double f(const std::vector<double>  &v);
double g(const std::vector<double>  &v);
```

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

- Modern C++ offers easy and type-safe interface to concurrency
- Low and high level programming tools provided

More Information

Further information

- **The Free Lunch is Over**
  - The Free Lunch Is Over, Herb Sutter, December 2004
  - Welcome to the Jungle, Herb Sutter, December 2011
  - Why C++?, Herb Sutter, C++ and Beyond 2011, Banff, Canada, August 2011
- **C++11/C++14**
  - C++ Working Draft, The first post-C++11 Working Paper, "The only chages since [C++11] are editorial“
  - isocpp.org
  - Online C++ reference
  - C++11, Wikipedia
  - Going Native 2012, Redmond, WA, February 2-3, 2012
  - Going Native 2013, Redmond, WA, September 4-6, 2013

Questions?

Thank you

Happy coding!!!

Backup

C++03 template metaprogramming example

C++03

```
// logic operations helper classes
struct false_type {
  typedef false_type type;
  enum { value = 0 };
};

struct true_type {
  typedef true_type type;
  enum { value = 1 };
};

// compile-time if statement
template<bool condition, class T, class U>
struct if_ { typedef U type; };

template <class T, class U>
struct if_<true, T, U> { typedef T type; };

// is_prime helper class - does most of the work
template<size_t N, size_t c>
struct is_prime_impl
{
  typedef typename if_<(c*c > N),
    true_type,
    typename if_<(N % c == 0),
      false_type,
      is_prime_impl<N, c+1> >::type >::type type;
  enum { value = type::value };
};
```

C++03

```
// is_prime user interface
template<size_t N>
struct is_prime
{
  enum { value = is_prime_impl<N, 2>::type::value };
};

// is_prime special cases
template <>
struct is_prime<0>
{
  enum { value = 0 };
};

template <>
struct is_prime<1>
{
  enum { value = 0 };
};

// Usage example
std::cout << is_prime<4256233>::value << std::endl;
```

Resource Acquisition Is Initialization (RAII)

```
class MyResource {
  // private resource
public:
  MyResource(/* args */)
  { // do whatever is needed to obtain ownership over resource }
  ~MyResource()
  { // do whatever is needed to reclaim the resource }
  // more class stuff here
};
```

*If you dissect the words of the **RAII** acronym (Resource Acquisition Is Initialization), you will think RAII is about acquiring resources during initialization. However the power of RAII comes not from tying **acquisition** to **initialization**, but from tying **reclamation** to **destruction**.*

Bjarne Stroustrup