Implicit Operations

Optimized C++

Ed Keenan

6 October 2019

13.0.6.3.18 Mayan Long Count

DePaul University

Goals

- Expose implicit behaviors
  - Write code to understand
- More Guidelines
  - Switches
  - Variables
- Compiler settings
  - Yes have a few dozen…
- This material isn't useful for social gathering
  - May put the listener to sleep

Dogs vs Cats

Who would win?

Implicit conversions

- Conversions:
  - ***int*** to ***float***
  - No biggie
  - Or is it true…
- On the XBox360 the penalty is very big.
  - from 1-4 cycles
  - to 40 cycles on XBox360
    - Don’t quote me, I’m going from memory.
- CPU can only do conversions on Read from main memory
  - Problem is, the data is discovered to be incorrect in the processor.

Implicit conversions

- How do we prevent implicit conversions from happening?
  - Turn up the compiler warnings
    - It notifies programmer
  - Treat Warning as errors
    - Great, but can’t guarantee engineers from turning it off.
  - What if you can make it a compiler error?
    - That would work independent of compilers setting.

Prevent implicit conversions

- How to prevent the compiler from calling the default constructor?

```
class dog()
{
   public:
            …
   private:
            dog();
};
```

Prevent implicit conversions

```
class dog
{
    public:
        void setX( float );
    private:
        void setX( int );
}
```

- This will prevent an ***int*** from being converted to a ***float***
- Gives a compiler error?
  - ***Great!***

Other inputs?

- **What happens if you passed in a:**
  - char?
  - double?
  - Apple that has an conversion operator to a float.
  - And so on?

- **How would you change this to handle anything?**

```
class dog
    {
        public:
            void setX( float );
        private:
            void setX( int );
    }
```

Templates to the rescue

```
class dog
{
public:
    void setX( float );
private:
    template <typename T> void setX( T );
}
```

- Why does it work?
  - Compiler matching rules
    - Everything matches template,
    - floats match the public method better (more restrictive)

- Templates provide the wild card options
  - Everything matches the template
  - Apples, oranges, char, ints…

Switch statements

- Always use enumerations, not raw ints.
  - Gives compiler hints on the range it can expect.
- Keep range close together.
  - Switch statements can become jump tables when cases are close together
  - If you have 50 cases, keep the enums 0-49 or 30000-30049

- Worse case it’s many if-else-if statements.
  - Place the most frequent cases first
  - Essentially it’s an early out.

Switch statements

- Graduate students:
  - In SE456 – Architecture of Real-time Systems
    - Sub for SE 450
  - Design Patterns – on steroids
    - Makes space invaders using modern design concepts
- Under Grad
  - In GAM 372 - Object-Oriented Game Development
  - Makes centipede using design patterns
- We learn about an alternative to switch statements with double dispatch

Variables usage

- Minimize local variables
  - Smaller the variable count, more opportunity to keep variables in registers
    - Good ☺
- Declare local variables in the inner most scope
  - Variables creation is delayed until it’s in scope.
  - External scope, may cause variable not to be called at all
    - Saving construction and destruction

Compiler settings

- Yeah-Boo
  - Performance vs Size tradeoffs
  - Debugging vs Speed
  - Compile time duration
- You can spend an eternity on this stuff
  - Many night go by trying and experimenting stuff
  - Need good test-bed first.
    - Memory manager perhaps…
- You can easily see 2-20 % savings depending on the setting
  - More you understand the better you can exploit
    - Are you using RTTI?

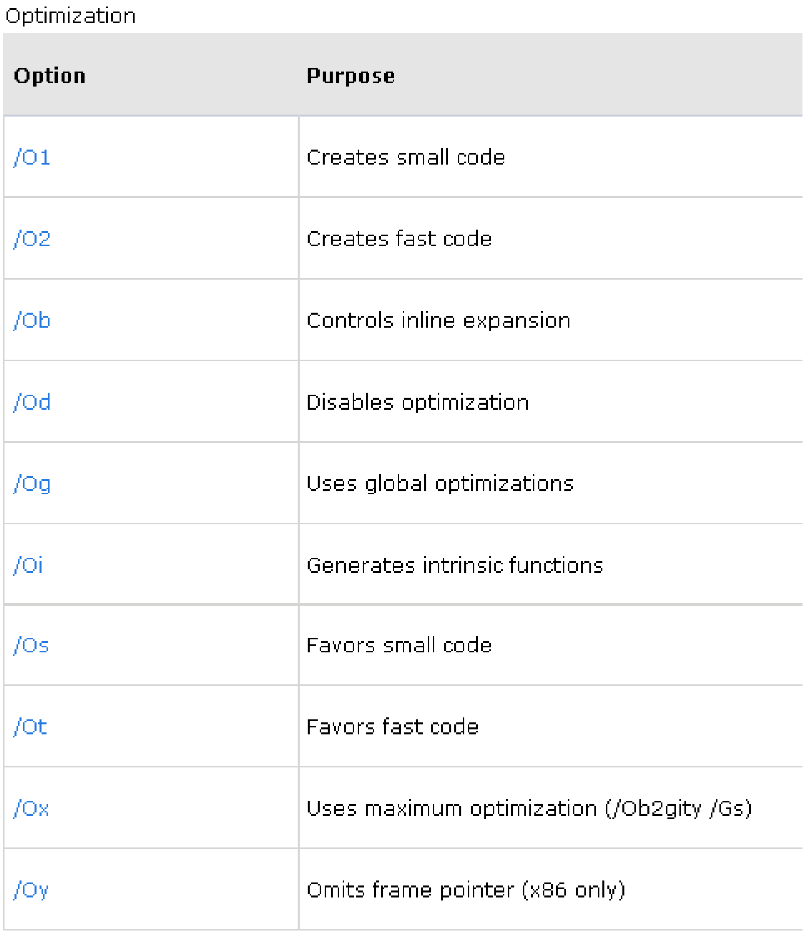
Compiler settings

- You know your environment
  - Compilers do not
- Different compilers perform differently
  - MS, Intel, GCC, Metroworks
  - Intel tends to be best on intel processors…
    - Go figure…
- Cheapest way to speed up your code
  - Change your settings
- War story
  - Triangle Strips

Microsoft Visual Studio

Compiler settings

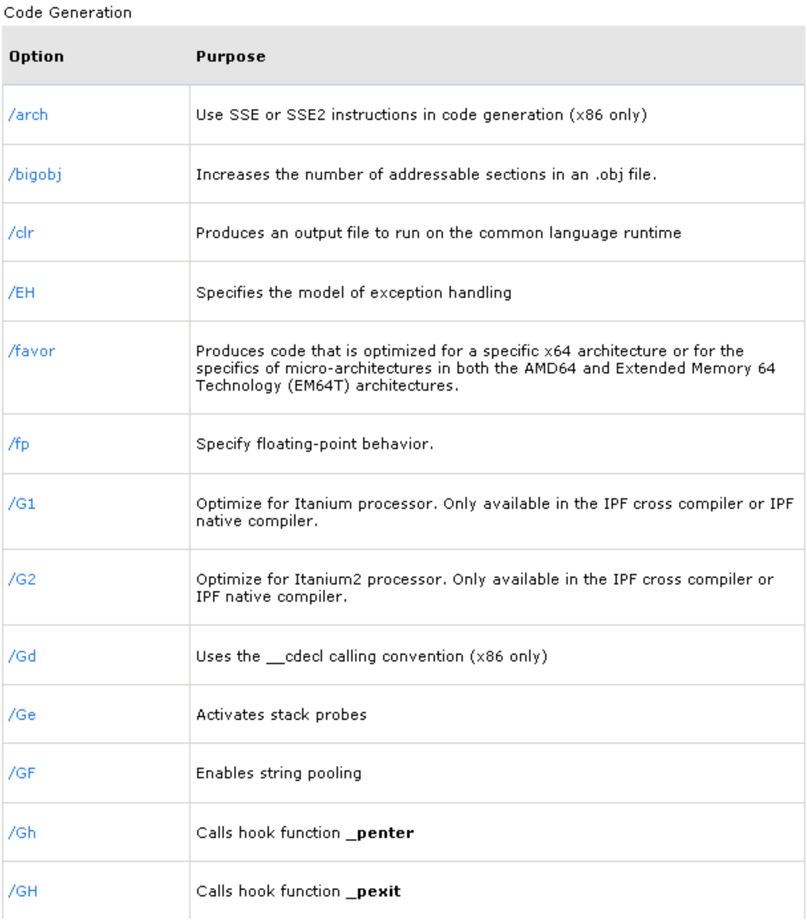
- Links to web pages
- VS 2017
  - https://docs.microsoft.com/en-us/cpp/what-s-new-for-visual-cpp-in-visual-studio
- VS 2018
  - https://docs.microsoft.com/en-us/cpp/build/reference/compiler-options



Optimization

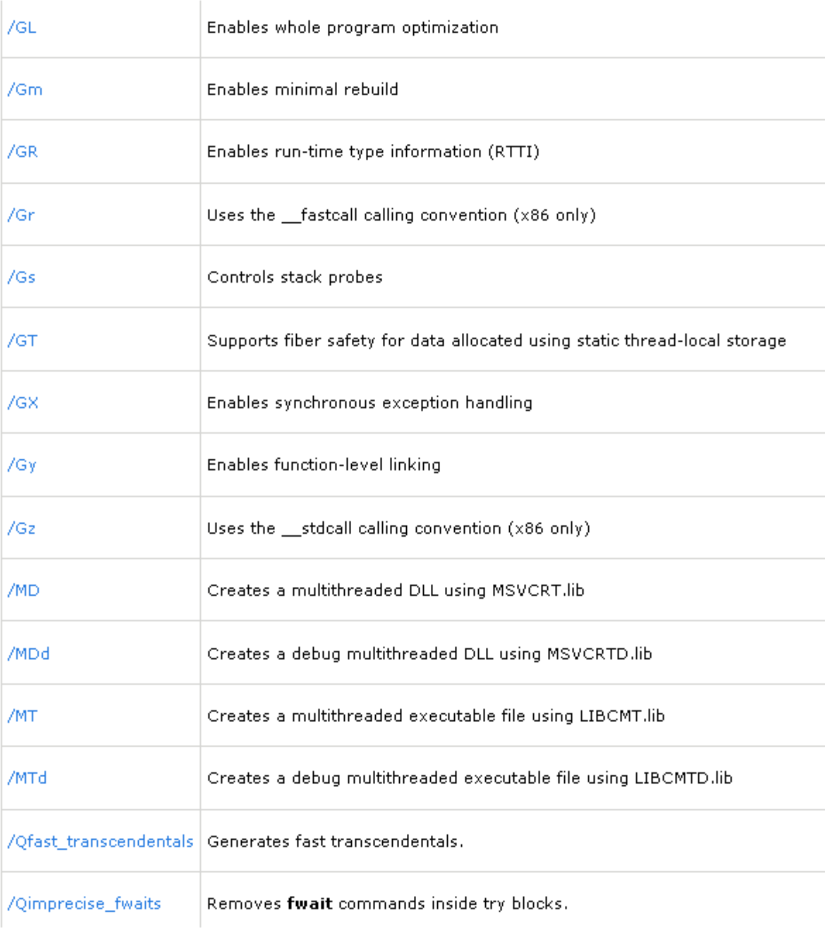
Option	Purpose
/O1	Creates small code
/O2	Creates fast code
/Ob	Controls inline expansion
/Od	Disables optimization
/Og	Uses global optimizations
/Oi	Generates intrinsic functions
/Os	Favors small code
/Ot	Favors fast code
/Ox	Uses maximum optimization (/Ob2gity /Gs)
/Oy	Omits frame pointer (x86 only)

Code Generation - settings



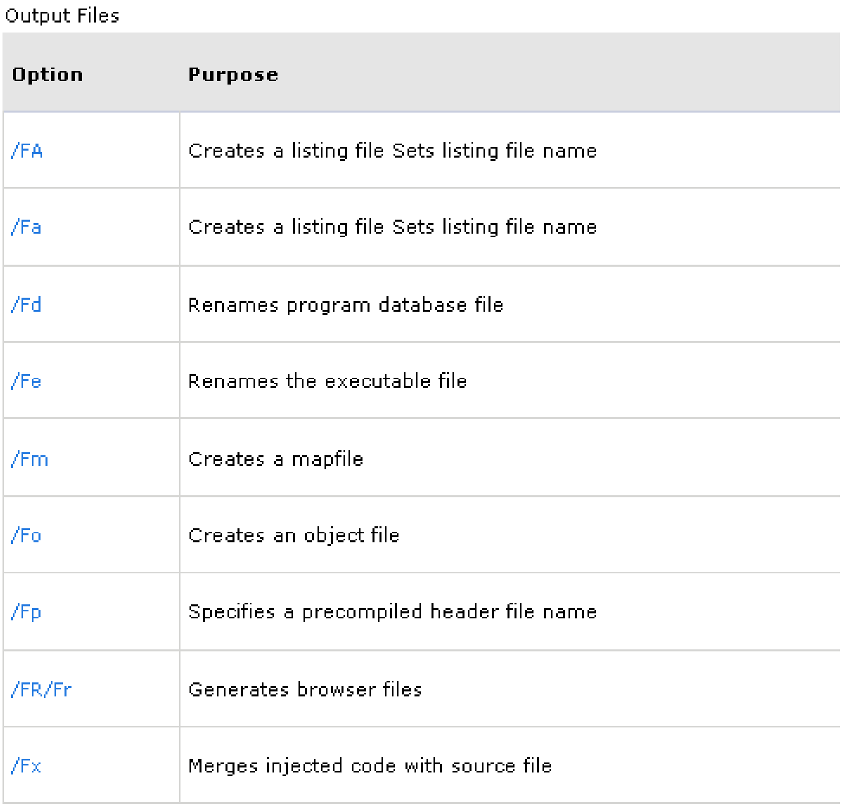
Code Generation

Option	Purpose
/arch	Use SSE or SSE2 instructions in code generation (x86 only)
/bigobj	Increases the number of addressable sections in an .obj file.
/clr	Produces an output file to run on the common language runtime
/EH	Specifies the model of exception handling
/favor	Produces code that is optimized for a specific x64 architecture or for the specifics of micro-architectures in both the AMD64 and Extended Memory 64 Technology (EM64T) architectures.
/fp	Specify floating-point behavior.
/G1	Optimize for Itanium processor. Only available in the IPF cross compiler or IPF native compiler.
/G2	Optimize for Itanium2 processor. Only available in the IPF cross compiler or IPF native compiler.
/Gd	Uses the __cdecl calling convention (x86 only)
/Ge	Activates stack probes
/GF	Enables string pooling
/Gh	Calls hook function **_penter**
/GH	Calls hook function **_pexit**



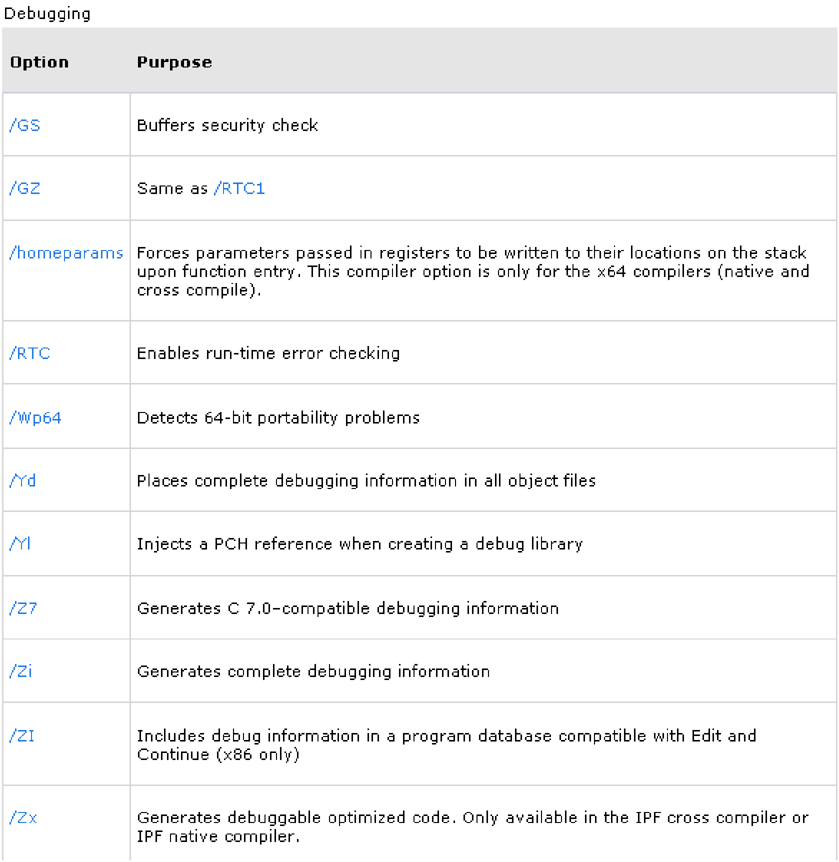
/GL	Enables whole program optimization
/Gm	Enables minimal rebuild
/GR	Enables run-time type information (RTTI)
/Gr	Uses the __fastcall calling convention (x86 only)
/Gs	Controls stack probes
/GT	Supports fiber safety for data allocated using static thread-local storage
/GX	Enables synchronous exception handling
/Gy	Enables function-level linking
/Gz	Uses the __stdcall calling convention (x86 only)
/MD	Creates a multithreaded DLL using MSVCRT.lib
/MDd	Creates a debug multithreaded DLL using MSVCRTD.lib
/MT	Creates a multithreaded executable file using LIBCMT.lib
/MTd	Creates a debug multithreaded executable file using LIBCMTD.lib
/Qfast_transcendentals	Generates fast transcendentals.
/Qimprecise_fwaits	Removes **fwait** commands inside try blocks.

Output & Debugging - settings



Output Files

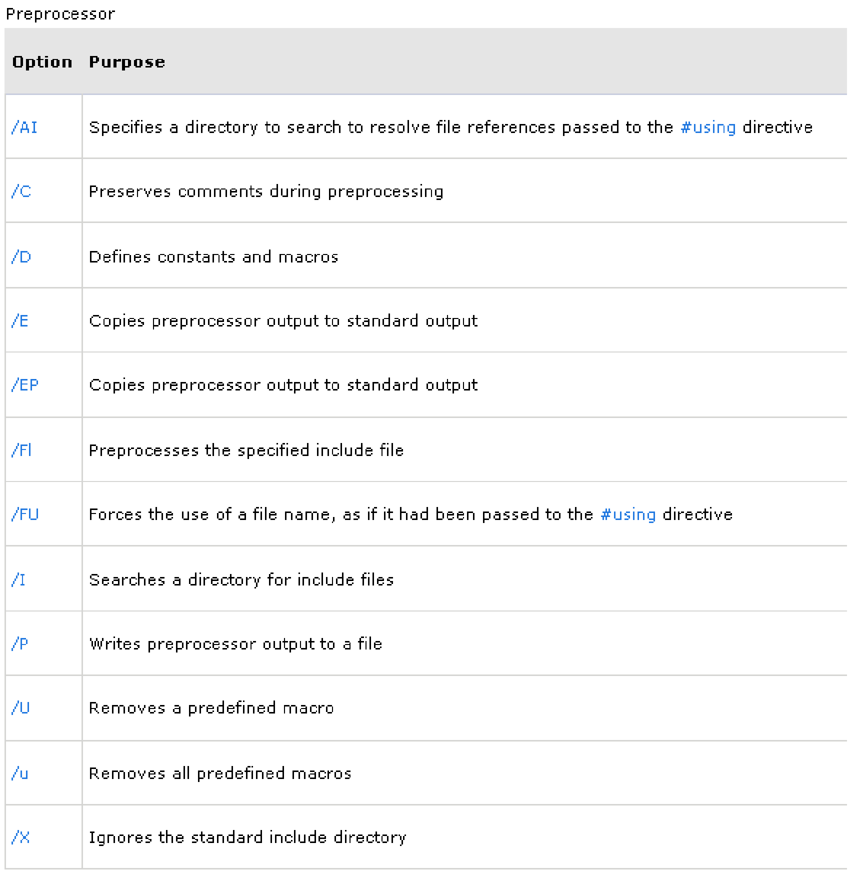
Option	Purpose
/FA	Creates a listing file Sets listing file name
/Fa	Creates a listing file Sets listing file name
/Fd	Renames program database file
/Fe	Renames the executable file
/Fm	Creates a mapfile
/Fo	Creates an object file
/Fp	Specifies a precompiled header file name
/FR/Fr	Generates browser files
/Fx	Merges injected code with source file



Debugging

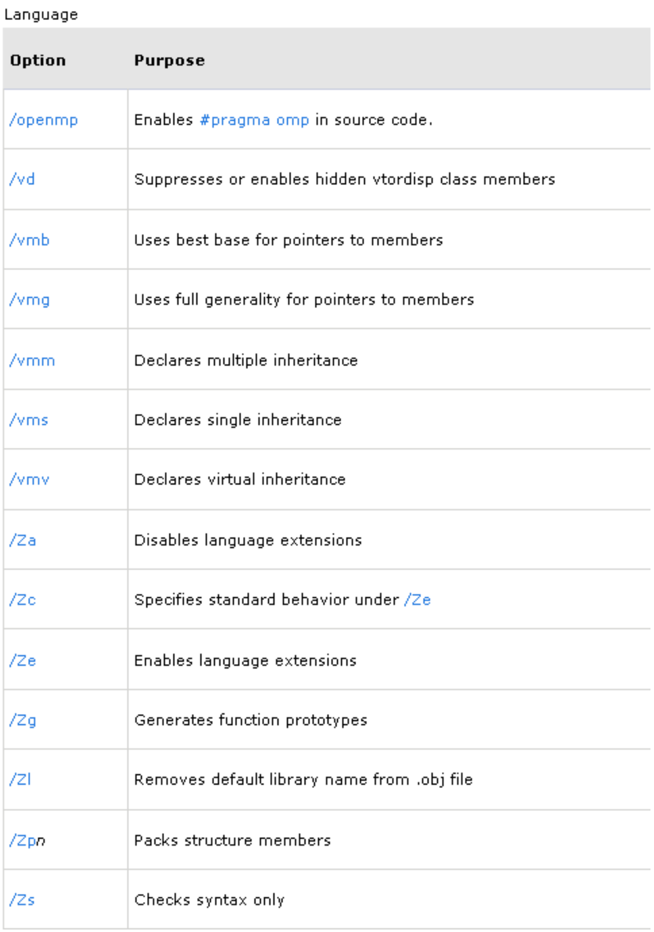
Option	Purpose
/GS	Buffers security check
/GZ	Same as /RTC1
/homeparams	Forces parameters passed in registers to be written to their locations on the stack upon function entry. This compiler option is only for the x64 compilers (native and cross compile).
/RTC	Enables run-time error checking
/Wp64	Detects 64-bit portability problems
/Yd	Places complete debugging information in all object files
/Yl	Injects a PCH reference when creating a debug library
/Z7	Generates C 7.0-compatible debugging information
/Zi	Generates complete debugging information
/ZI	Includes debug information in a program database compatible with Edit and Continue (x86 only)
/Zx	Generates debuggable optimized code. Only available in the IPF cross compiler or IPF native compiler.

Preprocessor / Language



Preprocessor

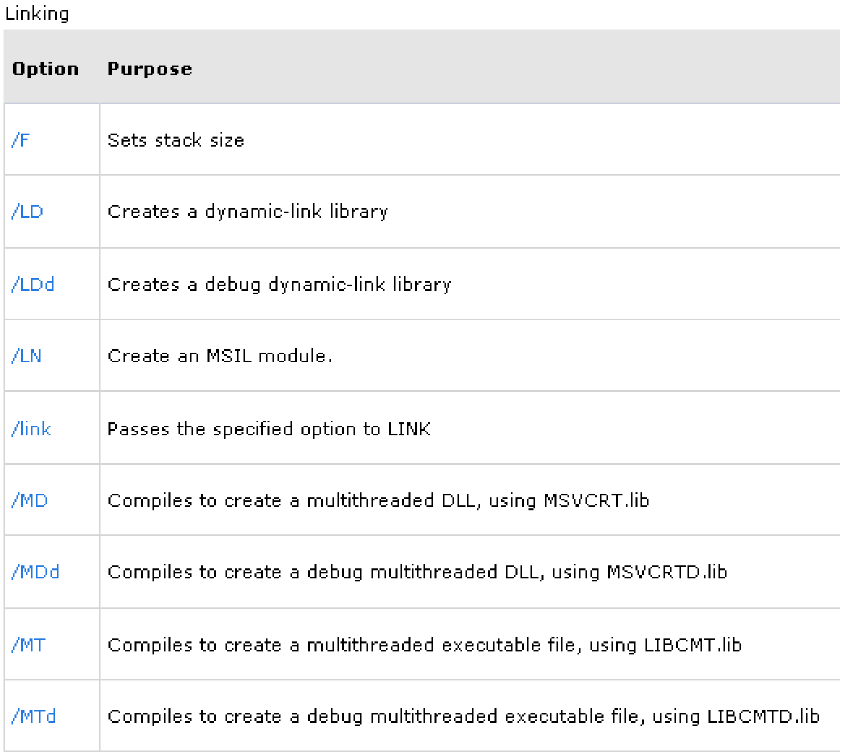
Option	Purpose
/AI	Specifies a directory to search to resolve file references passed to the #using directive
/C	Preserves comments during preprocessing
/D	Defines constants and macros
/E	Copies preprocessor output to standard output
/EP	Copies preprocessor output to standard output
/FI	Preprocesses the specified include file
/FU	Forces the use of a file name, as if it had been passed to the #using directive
/I	Searches a directory for include files
/P	Writes preprocessor output to a file
/U	Removes a predefined macro
/u	Removes all predefined macros
/X	Ignores the standard include directory



Language

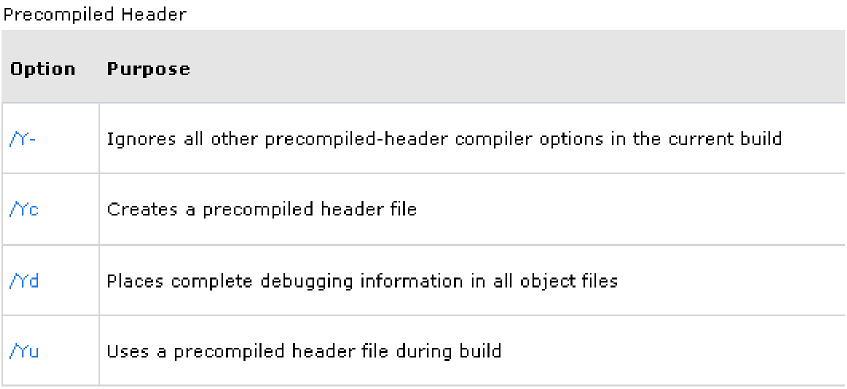
Option	Purpose
/openmp	Enables #pragma omp in source code.
/vd	Suppresses or enables hidden vtordisp class members
/vmb	Uses best base for pointers to members
/vmg	Uses full generality for pointers to members
/vmm	Declares multiple inheritance
/vms	Declares single inheritance
/vmv	Declares virtual inheritance
/Za	Disables language extensions
/Zc	Specifies standard behavior under /Ze
/Ze	Enables language extensions
/Zg	Generates function prototypes
/Zl	Removes default library name from .obj file
/Zp*n*	Packs structure members
/Zs	Checks syntax only

Linking / Precompiled Hdr



Linking

Option	Purpose
/F	Sets stack size
/LD	Creates a dynamic-link library
/LDd	Creates a debug dynamic-link library
/LN	Create an MSIL module.
/link	Passes the specified option to LINK
/MD	Compiles to create a multithreaded DLL, using MSVCRT.lib
/MDd	Compiles to create a debug multithreaded DLL, using MSVCRTD.lib
/MT	Compiles to create a multithreaded executable file, using LIBCMT.lib
/MTd	Compiles to create a debug multithreaded executable file, using LIBCMTD.lib



Precompiled Header

Option	Purpose
/Y-	Ignores all other precompiled-header compiler options in the current build
/Yc	Creates a precompiled header file
/Yd	Places complete debugging information in all object files
/Yu	Uses a precompiled header file during build

Thank You!

- Questions?

Return Value Optimization

Optimized C++

Ed Keenan

DePaul University

Goals

- Expose temporaries behaviors
  - Write code to understand
- Return Value Optimization
  - Yes, I'll take 2 bar keep.
- Alligators
  - Friend or Foe
    - Think like an Alligator
    - Live like an Alligator
    - Be an Alligator
- As always,
  - Great at social gatherings
    - Commencement ceremonies
    - Weddings
    - Funerals

It puts the ***fun*** back in ***funeral!***

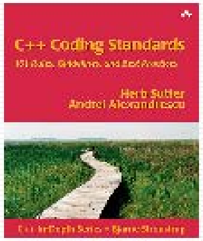
What’s going on?

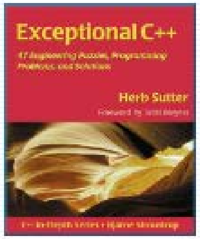
- What is the difference, if any, between the following?

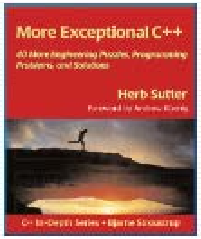
1. SomeType t = u;
2. SomeType t(u);
3. SomeType t();
4. SomeType t;

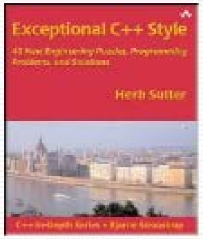
- SomeType t;
  - The variable t is initialised using the default ctor SomeType::SomeType().
- SomeType t();
  - This was a trick; it might look like a variable declaration, but it's a function declaration for a function t that takes no parameters and returns a SomeType.
- SomeType t(u);
  - This is direct initialisation. The variable t is initialised using SomeType::SomeType(u).
- SomeType t = u;
  - This is copy initialisation, and the variable t is always initialised using SomeType's copy ctor. (Even though there's an "=" there, that's just a syntax holdover from C... this is always initialisation, never assignment, and so operator= is never called.)

Guru of the week

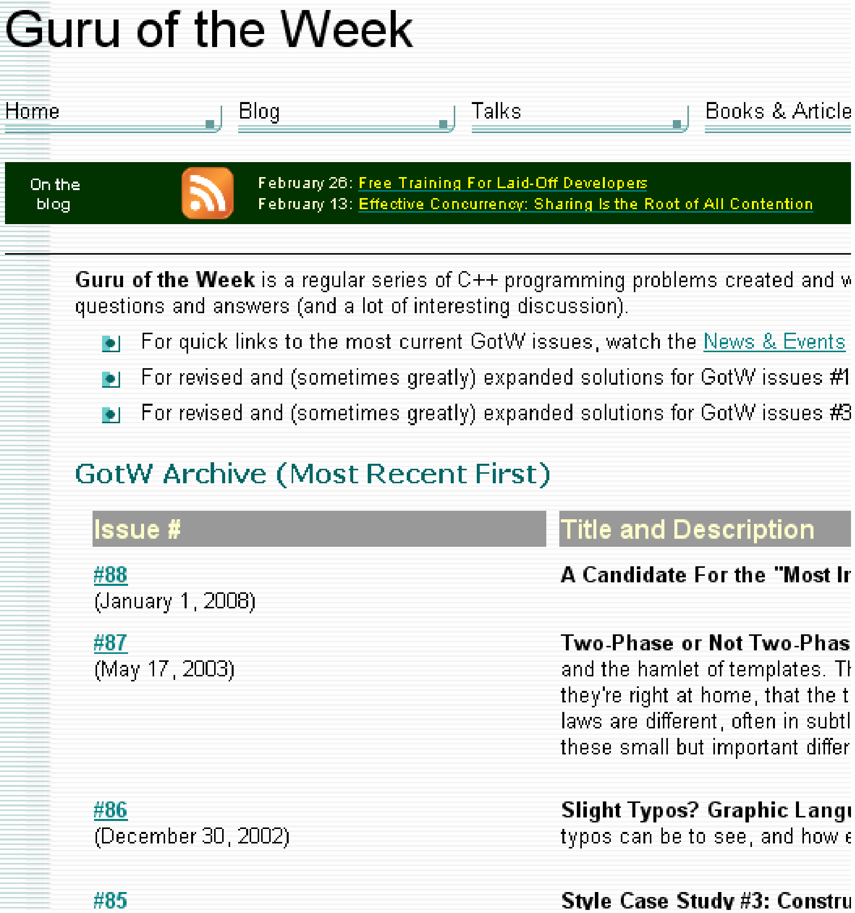

C++ Coding Standards
101 Rules, Guidelines, and Best Practices
Herb Sutter
Andrei Alexandrescu
C++ In-Depth Series • Bjarne Stroustrup


Exceptional C++
47 Engineering Puzzles, Programming Problems, and Solutions
Herb Sutter
C++ In-Depth Series • Bjarne Stroustrup


More Exceptional C++
40 New Engineering Puzzles, Programming Problems, and Solutions
Herb Sutter
C++ In-Depth Series • Bjarne Stroustrup


Exceptional C++ Style
40 New Engineering Puzzles, Programming Problems, and Solutions
Herb Sutter
C++ In-Depth Series • Bjarne Stroustrup

- Great website:
  - http://www.gotw.ca/gotw/
- Herb Sutter
  - C++ Language standards ANSI
  - Head of compiler for Microsoft
- Written several books:
  - Exceptional C++
    H. Sutter, Addison-Wesley, 2000, ISBN 0-201-61562-2.
  - More Exceptional C++
    H. Sutter, Addison-Wesley, 2002, ISBN 0-201-70434-X.
  - Exceptional C++ Style
    H. Sutter, Addison-Wesley, 2004, ISBN 0-201-76042-8.
  - C++ Coding Standards
    H. Sutter and A. Alexandrescu, Addison-Wesley, 2005, ISBN 0-321-11358-6.



Guru of the Week

February 26: Free Training For Laid-Off Developers
February 13: Effective Concurrency: Sharing Is the Root of All Contention

Guru of the Week is a regular series of C++ programming problems created and w
questions and answers (and a lot of interesting discussion).

- For quick links to the most current GotW issues, watch the News & Events
- For revised and (sometimes greatly) expanded solutions for GotW issues #1
- For revised and (sometimes greatly) expanded solutions for GotW issues #3

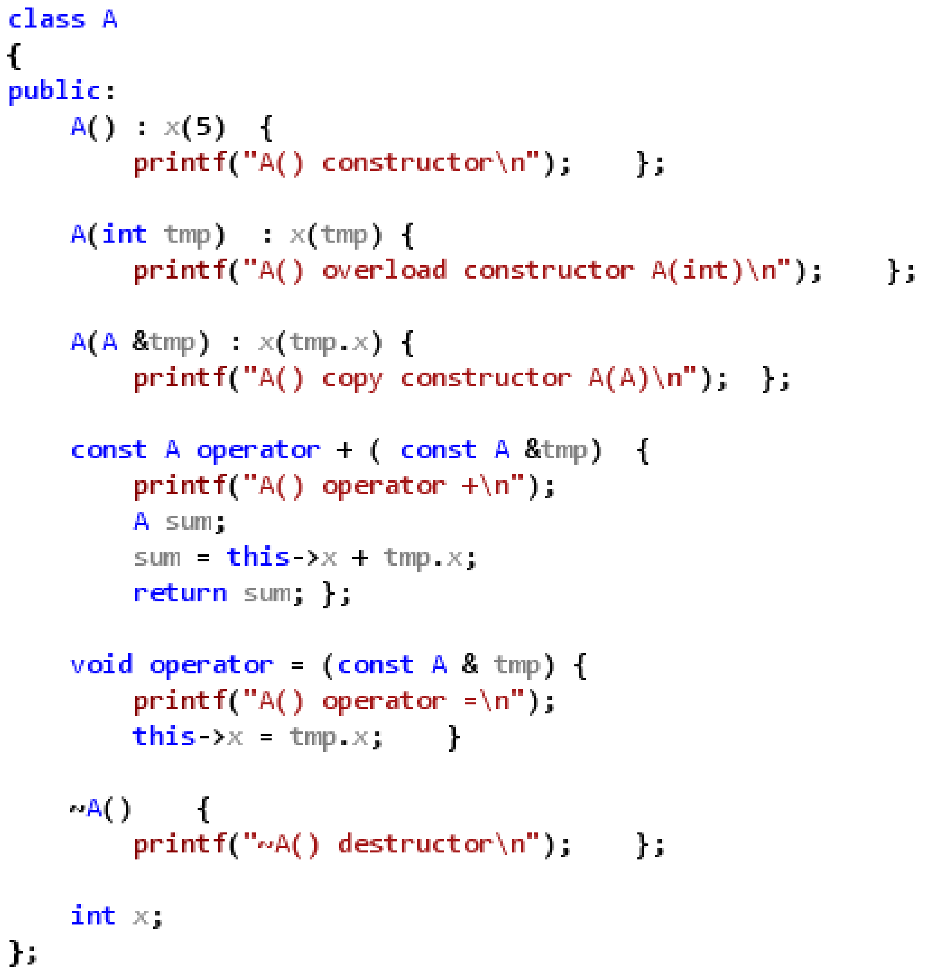
GotW Archive (Most Recent First)

Issue #	Title and Description
#88 (January 1, 2008)	**A Candidate For the "Most Ir**
#87 (May 17, 2003)	**Two-Phase or Not Two-Phas** and the hamlet of templates. Th they're right at home, that the t laws are different, often in subtl these small but important differ
#86 (December 30, 2002)	**Slight Typos? Graphic Langu** typos can be to see, and how e
#85	**Style Case Study #3: Constru**

DePaul University

Understand Implicit behavior

- Make a test bed
- Overload everything
- Add print statement everywhere
- It’s easy



```
class A
{
public:
    A() : x(5)  {
        printf("A() constructor\n");    };

    A(int tmp)  : x(tmp) {
        printf("A() overload constructor A(int)\n");    };

    A(A &tmp) : x(tmp.x) {
        printf("A() copy constructor A(A)\n");  };

    const A operator + ( const A &tmp)  {
        printf("A() operator +\n");
        A sum;
        sum = this->x + tmp.x;
        return sum; };

    void operator = (const A & tmp) {
        printf("A() operator =\n");
        this->x = tmp.x;    }

    ~A()    {
        printf("~A() destructor\n");    };

    int x;
};
```

Do it

Quiz

A tmp;

1. A() constructor
2. ~A() destructor

A tmp = 8;

1. A() overload constructor A(int)
2. ~A() destructor

A B;

A tmp(B);

1. B: A() constructor
2. tmp: A() copy constructor A(A)
3. tmp: ~A() destructor
4. B: ~A() destructor

A tmp(8);

1. A() overload constructor A(int)
2. ~A() destructor

- Sometimes the implicit calls surprise you until you print them out.
- Gives you valuable hints
  - Never ***Assume*** you understand the code.
  - ***ASS*** U ***ME***
- Sutter, Meyers
  - Consider their knowledge a 7/10
  - Language is that big and complex

Simple example?

(constructors/destructors)

1. A r(2);
2. A s(9);
3. A k;

4. k = s + r;

```
const  A  operator + ( const A &tmp)
{
      A sum;
      sum = this->x + tmp.x;
      return sum;
}
```

```
1.  r:  A: overload constructor A(int)
2.  s:  A: overload constructor A(int)
3.  k:  A() constructor
4.  s:  A: operator +
5.     --T1 Sum:  A: constructor
6.          --T2:  A: overload constructor A(int)
7.               --T1 Sum:   A: operator =
8.          --T2:  ~A() destructor
9.     --T3:  A: copy constructor A(A)
10.    --T1 Sum:  ~A() destructor
11. k:  A: operator =
12.    --T3:  ~A() destructor
13. k:  ~A() destructor
14. s:  ~A() destructor
15. r:  ~A() destructor
```

- 6 Constructors
- 6 Destructors

What happen?

- **Temporaries kill performance!**
- **Temporaries are Alligators.**
  - Sleeping,
  - hiding,
  - waiting to Bite you

You mean ***BYTE***

- **We added 3 temporaries**
  - T1, T2, T3
- **Is it possible to remove those temporaries?**
  - Yes – good news

Return Value Optimization (RVO)

- **Name Return Value Optimization (NRVO)**

```
const  A  operator + ( const A &tmp)
{
        A sum;
        sum = this->x + tmp.x;
        return sum;
}
```

  - ***sum*** is the NRVO
  - Some compilers can support this
  - Don’t depend on it!

- **Return Value Optimization**
  - Uses an unnamed return value
  - ***Construction*** on return
- **Preconditions**
  - One return in a function
    - Multiple functions will ***prevent*** RVO
  - If you don’t define the copy constructor
    - You turn ***OFF*** the RVO

RVO applied

1. A r(2);
2. A s(9);
3. A k;

4. k = s + r;

```
const  A  operator + ( const A &tmp)
{
    return A(this->x + tmp.x);
}
```

1. r: A() overload constructor A(int)
2. s: A() overload constructor A(int)
3. k: A() constructor
4. s: A: operator +
5. --T1: A: overload constructor A(int)
6. k: A: operator =
7. --T1: ~A() destructor
8. k: ~A() destructor
9. s: ~A() destructor
10. r: ~A() destructor

- 4 Constructors
- 4 Destructors

We can do better

1. A r(2);
2. A s(9);
3. A k = s + r;

```
const  A  operator + ( const A &tmp)
{
    return A(this->x + tmp.x);
}
```

1. r: A() overload constructor A(int)
2. s: A() overload constructor A(int)
3. s: A: operator +
4. k: A: overload constructor A(int)
5. k: ~A() destructor
6. s: ~A() destructor
7. r: ~A() destructor

- 3 constructors
- 3 destructors

Golden Hammer… (let’s find nails)

- **Which is better?**
  - x = x+5;
  - x += 5;
- ***x = x+5;***
  - Creates a temporary
  - Then does the assign
- ***x += 5;***
  - Writes through to x,
  - No temporary

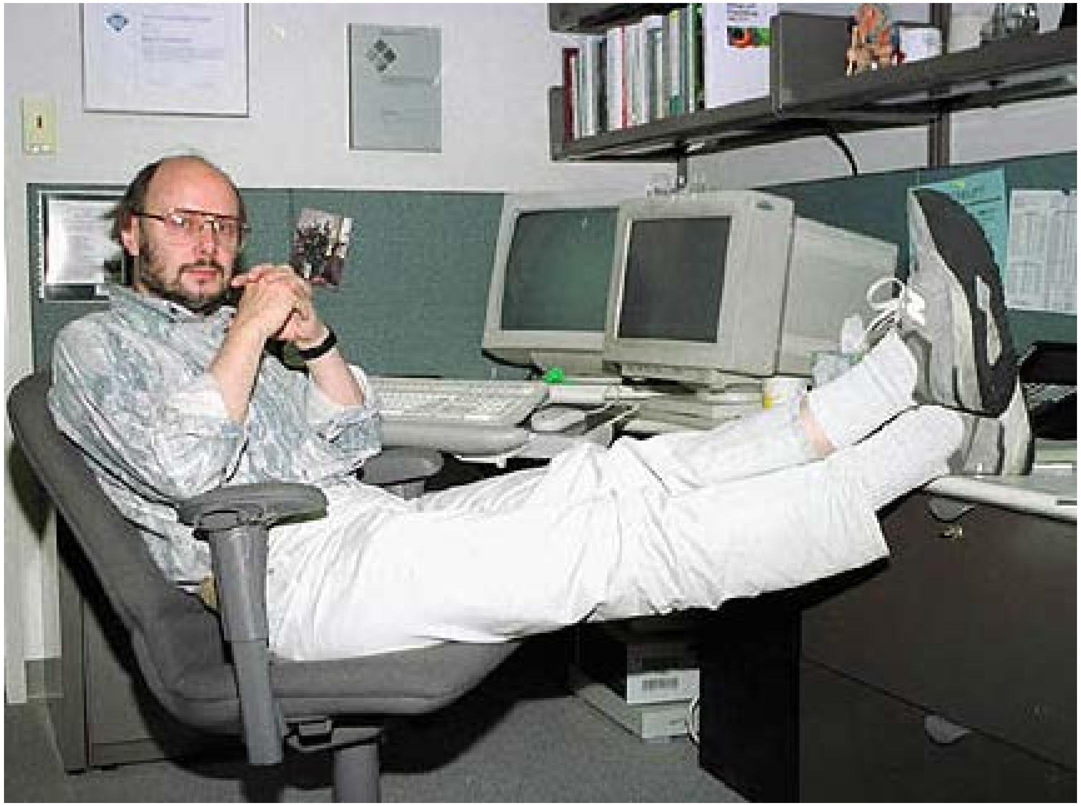
- **Same true for:**
  - +=
  - -=
  - /=
  - *=
- **Temporaries can be avoided another way.**
  - A = B + C
  - Rewrite as:
    - A = B;
    - A += C;

Questions

- On Board:

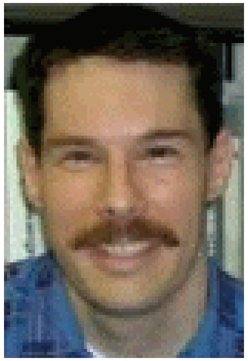
Name the Geek

(win a prize)



Bjarne Stroustrup
C++

With and without moustache



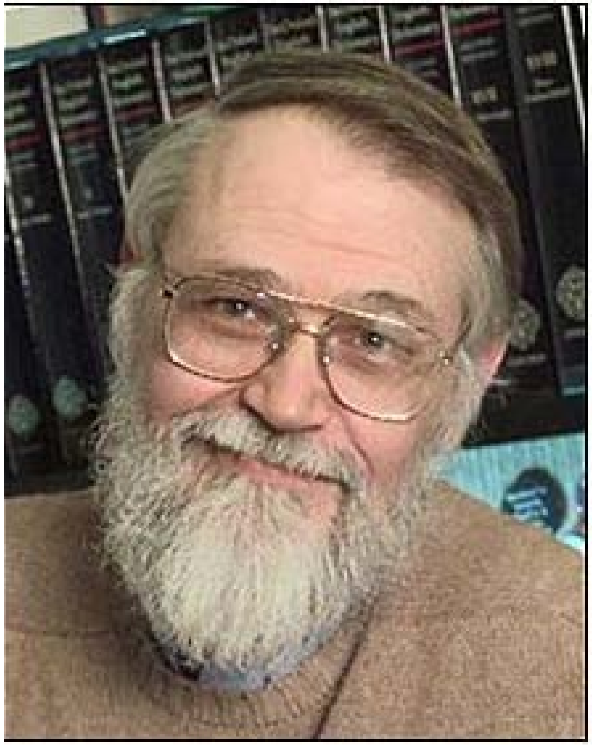
Herb Sutter
Compiler / C++ expert

Yes – Crazy hair guy

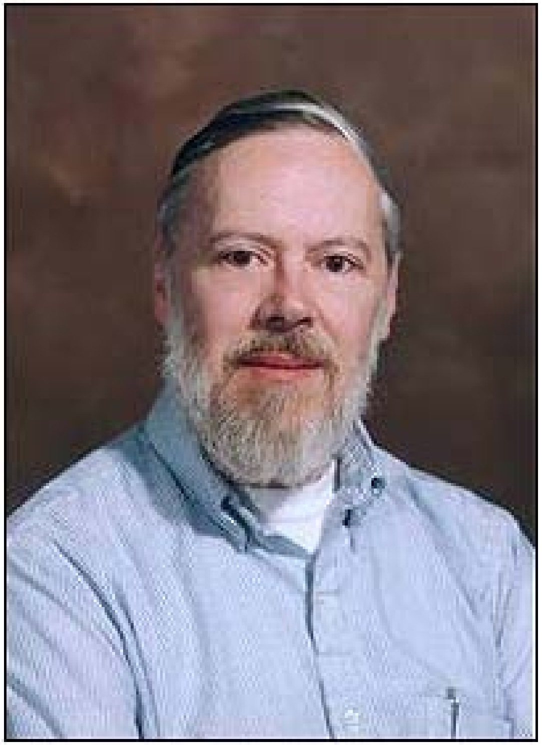
Scott Meyers
C++ Expert

Effective C++,STL
Series of books

& - is a hint



&



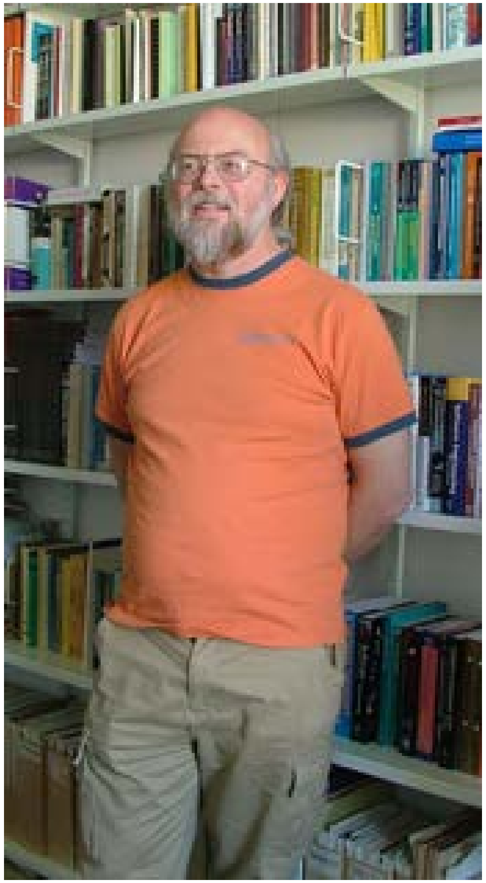
Brian Kernighan & Dennis Richie

K&R: The C programming language

Linus says thank your

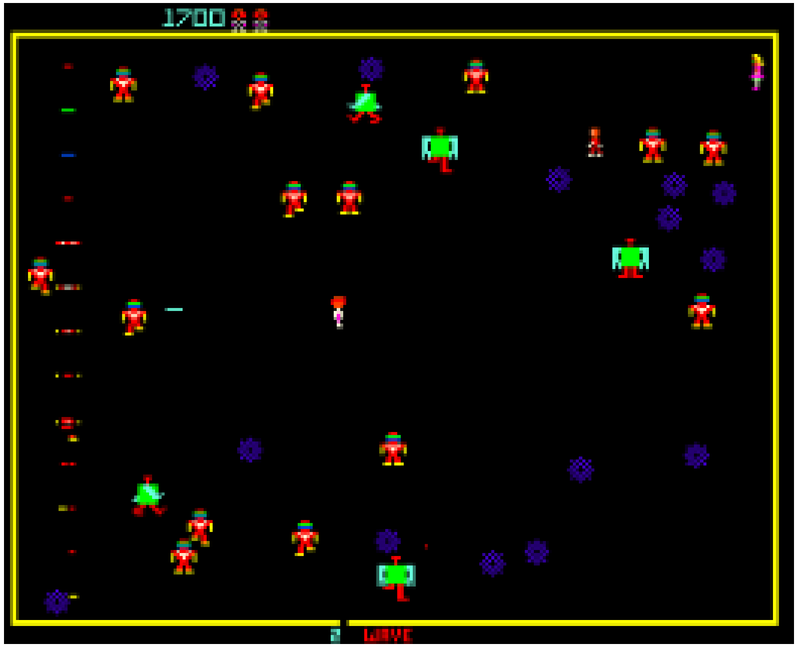
Ken Thompson
Unix

Likes Coffee



James Gosling
Java

Local celebrity



Eugene Jarvis
Robotron 2084 & Defender

Thank You!

- Questions?

Proxy Objects

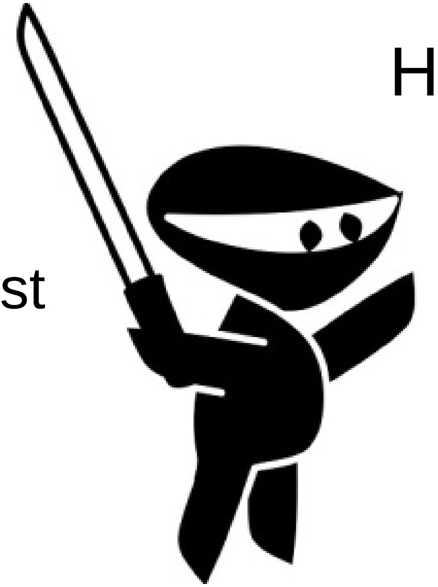
Optimized C++

Ed Keenan

DePaul University

Goals

- **Proxy objects**
  - They manipulate the CODE in cool ways
  - Sound smart
- **No practical use in optimizations**
  - Except for specialized tunneling statements into fast functions
  - Einstein and Bohr’s favorite C++ feature
- **Not useful at parties**
  - Only useful at laundry mats



Hey how about a movie or a theme?

Background story

- **I was reading the entire C++ language by Bjarne Stroustrup**
  - Yes I’m a Geek
- **At page 675**
  - 22.4.7 Temporaries, Copying, and Loops
    - I found this material
    - I was noodle mining
- **Lesson – read, learn, explore**
  - It pays off

- What is noodle mining?

- Video
  - Please.

Motivation

- Code:
  - Vect A,B,C,D,E;
  - E = A + B + C + D;
- From a temporary point of view:
  1. E = A + B + C + D;
  2. E = T1 + C + D;
  3. E = T2 + D;
  4. E = T3;

- To be optimal, need to follow C style function.
  - No temporaries
  - Copy no vectors
  - Minimal touch of components

```
void add4Vect(Vect &E, Vect &A, Vect &B,
                    Vect &C, Vect &D)
        {
        E.x = A.x + B.x + C.x + D.x;
        E.y = A.y + B.y + C.y + D.y;
        E.z = A.z + B.z + C.z + D.z;
        }
```

Constraints

- **How do we take advantage of C++ style and overloaded operators,**
  - while getting optimal performance interface?
- **What if the C++ code reduces down *automagically into the optimal interface?***
  - Only do this to function we care about.
  - Areas that the profiler targeted as heavy use operations.

Let's role

- Let's focus on 1st add
  - A+B

- Original function:

```
Vect  Vect::operator + (const Vect &tmp )
      {
      return Vect(x+tmp.x, y+tmp.y, z+tmp.z);
      }
```

- Instead of returning a Vect
  - Return a new Object
    - ***VaddV*** – Vect add Vect
    - Any name that you want

```
struct VaddV
{
      const Vect &v1;
      const Vect &v2;

      VaddV( const Vect &t1, const Vect &t2)
      : v1(t1), v2(t2) {};

      operator Vect()
      {
      return Vect(v1.x + v2.x, v1.y+v2.y, v1.z+v2.z);
      }
}

inline VaddV operator + (const Vect &a1,
                                        const Vect &a2)
      {
      return VaddV( a1, a2 );
      };
```

DEPAUL UNIVERSITY

Hold on, a little complex

Section A:

```
Vect  Vect::operator + (const Vect &tmp )
        {
        return Vect(x+tmp.x, y+tmp.y, z+tmp.z);
        }
```

Section B:

```
struct VaddV
{
        const Vect &v1;
        const Vect &v2;

    VaddV( const Vect &t1, const Vect &t2)
     : v1(t1), v2(t2) {};

        operator Vect()
        {
        return Vect(v1.x + v2.x, v1.y+v2.y, v1.z+v2.z);
        }
}
```

Section C:

```
inline VaddV operator + (const Vect &a1,
                             const Vect &a2)
        {
        return VaddV( a1, a2 );
        };
```

- E = A + B;
  - If there is only 2 variables,
  - Call **Vect**::operator + the original addition operator in **Vect**.
  - Call *Section A*
- E = A + B + C;
  - If there is >2

    &&
  - → **Vect**::operator + is ***NOT*** defined.
  - Call Section C
    - VaddV +

E = A + B;

```
struct VaddV
{
        const Vect &v1;
        const Vect &v2;

      VaddV( const Vect &t1, const Vect &t2)
       : v1(t1), v2(t2) {};

        operator Vect()
        {
        return Vect(v1.x + v2.x, v1.y+v2.y, v1.z+v2.z);
        }
}

inline VaddV operator + (const Vect &a1,
                                        const Vect &a2)
        {
        return VaddV( a1, a2 );
        };
```

1. **VaddV** + operator() is called
   - It creates a **VaddV** structure

2. **VaddV** constructor is called from the body
   - **VaddV** is instantiated

3. If the function needs to evaluate to a **Vect**
   - The operator **VaddV**::Vect() resolves the conversion from **VaddV** to **Vect**.

E = A + B + C;

```
struct  VaddV
{     const Vect &v1;
      const Vect &v2;

     VaddV( const Vect &t1, const Vect &t2)
      : v1(t1), v2(t2) {};

      operator Vect()
      {
      return Vect( v1.x + v2.x,
                   v1.y + v2.y,
                   v1.z + v2.z);
      }
}

inline VaddV operator + (const Vect &a1,
                                  const Vect &a2)
      {
      return VaddV( a1, a2 );
      };
```

```
struct VaddVaddV
{
      const Vect &v1;
      const Vect &v2;
      const Vect &v3;


     VaddVaddV( const VaddV &t1, const Vect &t2)
      : v1(t1.v1), v2(t1.v2), v3(t2) {};

      operator Vect()
      {
      return Vect( v1.x + v2.x + v3.x,
                   v1.y + v2.y  + v3.y,
                   v1.z  + v2.z + v3.z);
      }
}

inline VaddVaddV operator + (const VaddV &t1,
                                       const Vect &t2)
      {
      return VaddVaddV( t1, t2 );
      };
```

Evaluation E = A + B + C;

- A + B
  - creates **VaddV**
- VaddV + C
  - (A+B) + C
  - Creates **VaddVaddV**
- **VaddVaddV** is converted to Vect
  - Vect() takes 3 parameters

E = A + B + C + D

1. A+B -> **VaddV**
2. (A+B) + C -> **VaddV** + C -> **VaddVaddV**
3. (A+B+C) + D -> **VaddVaddV** + D -> VaddVaddVaddV
4. (A+B+C+D) -> VaddVaddVaddV convert to **Vect**

- **Vect()** now takes 4 inputs and return 5th.
  - Vect() gets A, B, C, D from VaddVaddVaddV

Commentary

- **This is complicated but it's worth it.**
  - Removes temporaries
  - Removes copies
  - Touching only variables needed
- **Work is done in Vect()**
  - That's the function to optimize with better math
- **C++ operators**
  - End user still can use overloaded operators +,-,*,/
- **Can retro fit this to existing code,**
  - without changing the existing client's code

Commentary cont.

- **Works for mixed types,**
  - Doesn't have to be the same objects.
  - See Stroustrup's example
    - uses Matrix,Vectors mixed
- **Only do this to code that is called a lot**
  - Use it sparingly

Thank You!

- Questions?