Some Programming Myths Revisited

Patrice Roy
Patrice.Roy@USherbrooke.ca
CeFTI, Université de Sherbrooke
Patrice.Roy@clg.qc.ca
Collège Lionel-Groulx

Who am I?

- Father of five (four girls, one boy), ages 24 to 6
- Feeds and cleans up after a varying number of animals
  - Look for « Paws of Britannia » with your favorite search engine
- Used to write military flight simulator code, among other things
  - CAE Electronics Ltd
- Full-time teacher since1998
  - Collège Lionel-Groulx, Université de Sherbrooke
  - Works a lot with game programmers
- Incidentally, WG21 and WG23 member (although I've been really busy recently)
  - Involved in SG12 and SG14, among other study groups
  - Occasional WG21 secretary
- And so on…

Programming Myths?

Programming Myths?

- We have been taught, or we ourselves have taught, things that we took for granted as being “good practice” in programming

Programming Myths?

- We have been taught, or we ourselves have taught, things that we took for granted as being "good practice" in programming
- Such things often stem from the "wisdom of the ancients"…
- …and are in effect part of our "myths"

Programming Myths?

- We have been taught, or we ourselves have taught, things that we took for granted as being “good practice” in programming
- Such things often stem from the “wisdom of the ancients”…
- …and are in effect part of our “myths”
  - Computer science being young as sciences come, some of the “ancients” are still among us and thriving today, and we're *so* lucky to have them!
    - We take part in a fun and awesome scientific endeavor!

Programming Myths?

- However, being as grounded in the science-that-there-was as these recommendations are, our ideas have evolved, so have our programming languages

Programming Myths?

- However, being as grounded in the science-that-there-was as these recommendations are, our ideas have evolved, so have our programming languages
- It can be interesting to revisit some of these taken-for-granted ideas

Impact of the Modern C++ Perspective

- In C++, particularly in what some call “modern C++”, we find a language that is different enough from its forebears to make revisiting our “myths” interesting

Impact of the Modern C++ Perspective

- In C++, particularly in what some call "modern C++", we find a language that is different enough from its forebears to make revisiting our "myths" interesting
- How do such things as "goto considered harmful" or "only one return per function", for example, hold as "wisdom" with respect to modern C++?
  - Do they still help us write better programs?
  - Should be "rethink" them under the light of modern languages and practice?

Impact of the Modern C++ Perspective

- The aim of this talk is to examine what some commonly heard / commonly taught recommendations or advices with respect to programming practice mean in the context of "modern" C++

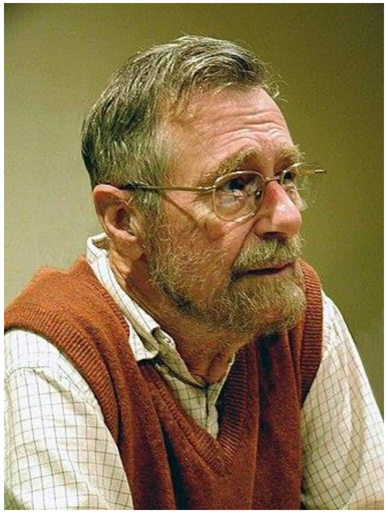
Impact of the Modern C++ Perspective

- The aim of this talk is to examine what some commonly heard / commonly taught recommendations or advices with respect to programming practice mean in the context of "modern" C++
- We will take a small set of such advices, present them in context, show how well (or how badly) they suit today's C++, and try to rephrase them if this seems advantageous

`goto` Considered Harmful

`goto` Considered Harmful

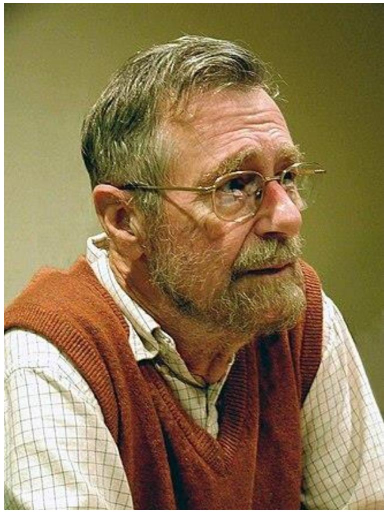
- We owe this formulation to Edsger W. Dijkstra
  - Go To Statement Considered Harmful, Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148.
    - Rather short paper, but what impact!



(taken from Wikipedia)

`goto` Considered Harmful

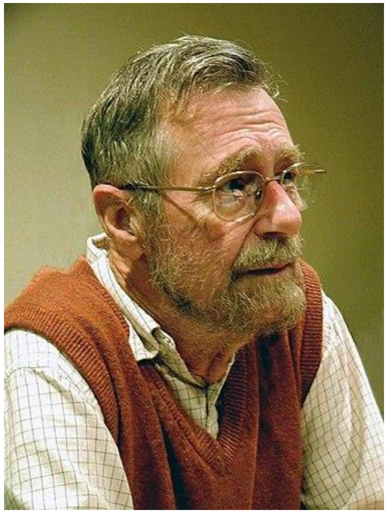
- In a note to the editor:
  - "*For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code) [. . .]*"



(taken from Wikipedia)

`goto` Considered Harmful

- In a note to the editor (my emphasis):
  - "*For a number of years I have been familiar with the observation* ***that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce****. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the* ***go to statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code)*** *[…]*"



(taken from Wikipedia)

`goto` Considered Harmful

- That seems reasonable, even for trivial programs

goto Considered Harmful

```
#include <iostream>
int main() {
    int sum = 0;
    int i = 1;
    while (i <= 10) {
        sum += i;
        ++i;
    }
    std::cout << sum; // 55
}
```

`goto` Considered Harmful

```
#include <iostream>
int main() {
   int sum = 0;
   int i = 1;
test:
   if (i > 10) goto end;
   sum += i;
   ++i;
   goto test;
end:
   std::cout << sum; // 55
}
```

`goto` Considered Harmful

- Same effect, *similar* generated code
  - Without goto: https://godbolt.org/z/9bwLhe
  - With goto: https://godbolt.org/z/xJxX49
- Note that writing a `goto`-based loop is "kind of" like trying to second-guess your compiler…
  - You might win some battles, but you're likely to lose most of them

`goto` Considered Harmful

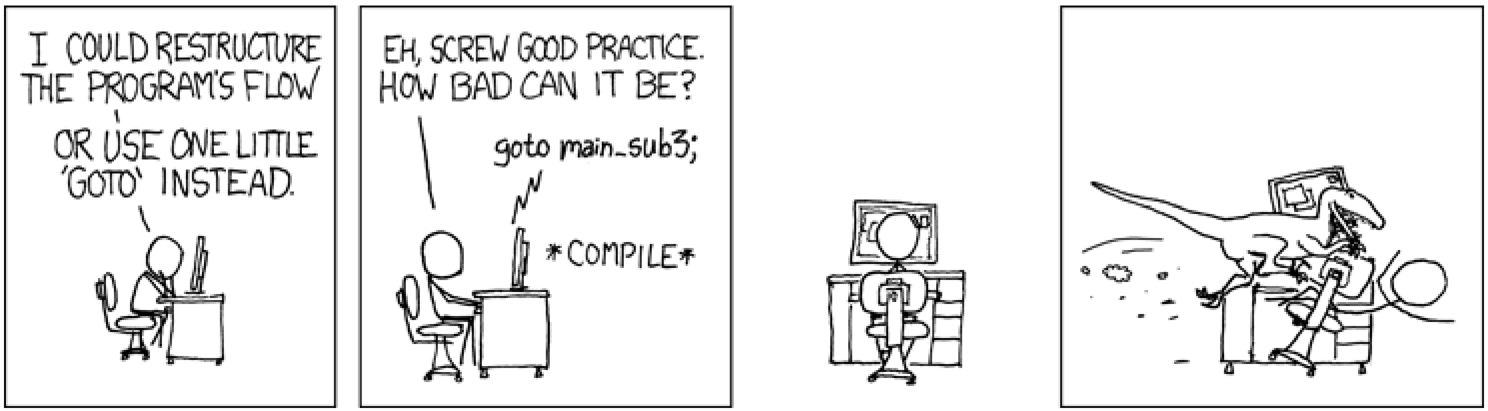
- One implementation (the `goto`-less one!) more directly expresses intent than the other
  - Debuggability
  - Teachability
  - Ease of maintenance
  - etc.
- Thinking at a higher level of abstraction is the key point of Dijkstra's recommendation
  - In C++, the `for`, `while`, `do`, `if`… are translated into the equivalent of `goto` statements, but bring more *structure* to source code
    - *Structured* programming!

`goto` Considered Harmful

- Everyone agreed…

`goto` Considered Harmful

- Everyone agreed…



http://imgs.xkcd.com/comics/goto.png (Randall Munroe)

`goto` Considered Harmful

- Everyone agreed… Right?

`goto` Considered Harmful

- Structured Programming with go to Statements, Donald E. Knuth
  - Computing Surveys, Vol. 6, Number 4, December 1974, pp.261-301. Copyright © 1974, Association for Computing Machinery, Inc.
    - Much bigger paper!

(taken from Wikipedia)

`goto` Considered Harmful

- From the introduction:
  - *“[…] This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without go to statements; (b) a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code. The discussion brings out opposing points of view about whether or not go to statements should be abolished; some merit is found on both sides of this question”*

(taken from Wikipedia)

`goto` Considered Harmful

- From the introduction (my emphasis):
  - *"[...] This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without go to statements; (b)* ***a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code****. The discussion brings out opposing points of view about* ***whether or not go to statements should be abolished; some merit is found on both sides of this question****"*

(taken from Wikipedia)

`goto` Considered Harmful

- Knuth brings more than one consideration to the discussion
  - Early exit from a function is one
    - We'll return to that later
  - Let's take a look at some of these considerations

`goto` Considered Harmful

- Knuth mentions the simplification of some algorithmic constructs
  - Today, we would probably examine his examples under the light of nesting reduction

`goto` Considered Harmful

```
// ... Lots of nesting
while(!done) {
   if (A) {
      // work that depends on A
      if(B) {
         // work that depends on A and B
         if(C) {
            // work that depends on A && B && C
         }
      }
   }
}
// ...
```

`goto` Considered Harmful

```
// ... More linear structure
test:
while(!done) {
    if (!A) goto test;
    // work that depends on A
    if (!B) goto test;
    // work that depends on A and B
    if (!C) goto test;
    // work that depends on A && B && C
}
// ...
```

`goto` Considered Harmful

```
// ... More linear structure (using
// more ... acceptable mechanisms?)
while(!done) {
    if (!A) continue;
    // work that depends on A
    if (!B) continue;
    // work that depends on A and B
    if (!C) continue;
    // work that depends on A && B && C
}
// ...
```

`goto` Considered Harmful

```
// ... Not necessarily a mechanical
// transformation
while(!done) {
   if (A) {
      X x0; // x0 ctor
      // work that depends on A
      if(B) {
         Y y0; // y0 ctor
         // work that depends on A and B
      } // y0 dtor
   } // x0 dtor
}
// ...
```

`goto` Considered Harmful

```
// ...
test:
while(!done) {
    if (!A) goto test;
    X x0; // x0 ctor
    // work that depends on A
    // maybe surprising : Ok, x0 dtor will
    // be called even if branch is taken
    if (!B) goto test;
    Y y0; // y0 ctor
    // work that depends on A and B
} // y0 dtor then x0 dtor
// ...
```

`goto` Considered Harmful

```
// ... This form of goto has a standard
// incarnation
while(!done) {
    if (!A) continue;
    X x0; // x0 ctor
    // work that depends on A
    if (!B) continue;
    Y y0; // y0 ctor
    // work that depends on A and B
} // y0 dtor then x0 dtor
// ...
```

`goto` Considered Harmful

```
#include <cstdio>
struct X {
   X() { puts("X::X()"); }
   ~X() { puts("X::~X()"); }
};
int main() {
   int i = 0;
here:
   X x0;
   if (++i < 5) goto here;
} // x0 ctor and x0 dtor called 5 times
```

`goto` Considered Harmful

```
// however, skipping construction is not allowed
// if initialization is non-vacuous
#include <cstdio>
#include <cstdlib>
using std::rand;
bool maybe_skip() { return rand() % 10 != 0; }
struct X {
   X() { puts("X::X()"); }
   ~X() { puts("X::~X()"); }
};
int main() {
   // illegal, might skip x0 ctor
   if (maybe_skip()) goto here;
   X x0;
here:
   ;
}
```

`goto` Considered Harmful

```
// however, skipping construction is not
// allowed if initialization is non-vacuous
// (so this one compiles... Sorry!)
#include <cstdlib>
using std::rand;
bool maybe_skip() {
   return rand() % 10 != 0;
}
int main() {
   if (maybe_skip()) goto here;
   int n; // potential warning
          // (variable not initialized)
here:
   ;
}
```

`goto` Considered Harmful

- Knuth was also interested in efficient ways to perform early exit from loops

`goto` Considered Harmful

```
// late exit from loop
template <class T, int N>
bool contains(T && x, const T (&arr)[N]) {
    // inefficient (goes through all
    // elements even when we know we
    // have our answer
    bool result = false;
    for(auto & obj : arr)
        if (obj == x)
            result = true;
    return result;
}
```

`goto` Considered Harmful

```
// early exit from loop (slow, painful)
template <class T, int N>
bool contains(T &&x, const T(&arr)[N]) {
    // also inefficient (three tests
    // per iteration for loop control)
    bool result = false;
    for(int i = 0; !result && i != N; ++i)
        if (arr[i] == x)
            result = true;
    return result;
}
```

`goto` Considered Harmful

```
// early exit from loop
template <class T, int N>
bool contains(T && x, const T (&arr)[N]) {
   // more efficient (could be better)
   bool result = false;
   for(auto & obj : arr)
      if (obj == x) {
         result = true;
         goto done;
      }
done:
   return result;
}
```

`goto` Considered Harmful

```
// early exit from loop (formalized)
template <class T, int N>
bool contains(T && x, const T (&arr)[N]) {
   // more efficient (could be better)
   bool result = false;
   for(auto & obj : arr)
      if (obj == x) {
         result = true;
         break;
      }
   return result;
}
```

`goto` Considered Harmful

- If your code uses `break` or `continue` in loops, then you're already using structured forms of `goto`
  - Of course, since there is a formalization of these code structures, use them: prefer `break` and `continue` to `goto` for such cases
    - Facilitates reasoning
    - Simplifies object lifetime management

`goto` Considered Harmful

- Knuth was also concerned about an important use case: **error handling**
  - In general, error handling tends to obscure code with artefacts not entirely relevant to "normal" processing
  - Knuth suggested to direct these unusual situations in some other location of the program
    - This is reminiscent of `on error goto…` statements such as those found in pre-.NET Visual Basic

`goto` Considered Harmful

- Paraphrased as C++, his example:

```
char memory[SIZE];
int cur = 0;
void *p;
// ... allocate a block of n bytes
if(cur + n >= SIZE) goto mem_overflow;
p = &memory[cur];
cur += n;
// ... use p ...
// ...
mem_overflow:
// ... handle out of memory error
```

`goto` Considered Harmful

- This can be seen as having been formalized through exception handling:

```
char memory[SIZE];
int cur = 0;
void *p;
// ... allocate a block of n bytes
try {
   if(cur + n >= SIZE) throw bad_alloc{};
   p = &memory[cur];
   cur += n;
   // ... use p ...
   // ...
} catch(bad_alloc&) {
   // ... handle out of memory error
}
```

`goto` Considered Harmful

- Another interesting use case of `goto` is… the `switch` statement
  - That form is even called `GOTO` in Fortran 77 (this is considered obsolete in Fortran 95)

`goto` Considered Harmful

- Another interesting use case of `goto` is… the `switch` statement
  - That form is even called `GOTO` in Fortran 77 (this is considered obsolete in Fortran 95)
- The switch statement in C++ does not have the same… *structured-programming-ness* as `if`, `while`, `for` or `do` statements

`goto` Considered Harmful

```
// simple factorial implementation
unsigned long long facto(int n) {
    auto fac = 1ULL;
    for(int i = 2; i <= n; ++i)
        fac *= i;
    return fac;
}
```

`goto` Considered Harmful

```
// manual loop unrolling inspired by Duff's Device
unsigned long long factorial(int n) {
   auto fac = 1ULL;
   if (n == 0) return 1;
   if (n % 4 == 0) fac *= n--;
   int i = 1;
   switch (4 - n % 4) {
      do {
   case 0: fac *= i++; [[fallthrough]];
   case 1: fac *= i++; [[fallthrough]];
   case 2: fac *= i++; [[fallthrough]];
   case 3: fac *= i++;
      } while (i <= n);
   }
   return fac;
}
```

`goto` Considered Harmful

```
// manual loop unrolling inspired by Duff's Device
unsigned long long factorial(int n) {
   auto fac = 1ULL;
   if (n == 0) return 1;
   if (n % 4 == 0) fac *= n--;
   int i = 1;
   switch (4 - n % 4) {
      do {
   case 0: fac *= i++; [[fallthrough]];
   case 1: fac *= i++; [[fallthrough]];
   case 2: fac *= i++; [[fallthrough]];
   case 3: fac *= i++;
      } while (i <= n);
   }
   return fac;
}
```

Suppose we call `factorial(5)`

`goto` Considered Harmful

```
// manual loop unrolling inspired by Duff's Device
unsigned long long factorial(int n) {
    auto fac = 1ULL;
    if (n == 0) return 1;
    if (n % 4 == 0) fac *= n--;
    int i = 1;
    switch (4 - n % 4) {
        do {
    case 0: fac *= i++; [[fallthrough]];
    case 1: fac *= i++; [[fallthrough]];
    case 2: fac *= i++; [[fallthrough]];
    case 3: fac *= i++;
        } while (i <= n);
    }
    return fac;
}
```

4 - (5 % 4) == 4 - 1
Thus we jump to case 3…

… where we do the first multiplication + incrementation

`goto` Considered Harmful

```
// manual loop unrolling inspired by Duff's Device
unsigned long long factorial(int n) {
   auto fac = 1ULL;
   if (n == 0) return 1;
   if (n % 4 == 0) fac *= n--;
   int i = 1;
   switch (4 - n % 4) {
      do {
   case 0: fac *= i++; [[fallthrough]];
   case 1: fac *= i++; [[fallthrough]];
   case 2: fac *= i++; [[fallthrough]];
   case 3: fac *= i++;
      } while (i <= n);
   }
   return fac;
}
```

… only then do we do the test, and only do once per four computations thereafter

`goto` Considered Harmful

- Amusingly (!), such tricks as Duff's Device cannot be done naturally in C# or Java
  - Relies on fallthrough behavior
  - In C# for example, `break` statements are mandatory…
    - … even in `default`!
    - (you can circumvent this issue… with a `goto` to the next label!)
- If you sometimes use `switch` statements in your code, you are in fact using `goto` statements

goto (

```
static void F(int n)
{
    switch(n % 3)
    {
    case 2: Console.WriteLine("2"); goto case 1;
    case 1: Console.WriteLine("1"); goto default;
    default: Console.WriteLine("0"); break;
    }
}
```

- Amusingly … ne
  naturally in
  - Relies on
  - In C# for example, … statements are mandatory…
    - … even in `default`!
    - (you can circumvent this issue… with a `goto` to the next label!)
- If you sometimes use `switch` statements in your code, you are in fact using `goto` statements

`goto` Considered Harmful

```
// compiles (probably with a warning)
struct X {
   X() { cout << "X::X()" << endl; }
   ~X() { cout << "X::~X()" << endl; }
};
int main() {
   srand(static_cast<
      unsigned int
   >(time(nullptr)));
   switch(rand() % 2) {
       X x; // warning : will never be reached
   }
}
```

`goto` Considered Harmful

```
// compiles (probably with a warning)
struct X {
   X() { cout << "X::X()" << endl; }
   ~X() { cout << "X::~X()" << endl; }
};
int main() {
   srand(static_cast<
      unsigned int
   >(time(nullptr)));
   switch(rand() % 2) {
       X x; break; // likewise
   }
}
```

`goto` Considered Harmful

```
// compiles just fine (I know...)
struct X {
   X() { cout << "X::X()" << endl; }
   ~X() { cout << "X::~X()" << endl; }
};
int main() {
   srand(static_cast<
      unsigned int
   >(time(nullptr)));
   switch(rand() % 2) {
   case 0:
       X x;
   }
}
```

`goto` Considered Harmful

```
// switch statements cannot "cross" object
// initialization
struct X {
   X() { cout << "X::X()" << endl; }
   ~X() { cout << "X::~X()" << endl; }
};
int main() {
   srand(static_cast<unsigned int>(time(nullptr)));
   switch(rand() % 2) {
   case 0:
       X x;
   case 1: // illegal (x would be accessible but not
       ;   // initialized)
   }
}
```

`goto` Considered Harmful

```
// switch statements cannot "cross" object
// initialization
struct X {
   X() { cout << "X::X()" << endl; }
   ~X() { cout << "X::~X()" << endl; }
};
int main() {
   srand(static_cast<unsigned int>(time(nullptr)));
   switch(rand() % 2) {
   case 0:
       X x; break; // still illegal (same reason)
   case 1:
       ; // x would exist and be reachable here
   }
}
```

goto Considered Harmful

```
// switch statements cannot "cross" object initialization
struct X {
   X() { cout << "X::X()" << endl; }
   ~X() { cout << "X::~X()" << endl; }
};
int main() {
   srand(static_cast<unsigned int>(time(nullptr)));
   switch(rand() % 2) {
   case 0:
   {
      X x;
   }
   case 1: // Ok (x's scope avoids the problem)
   }
}
```

`goto` Considered Harmful

- Dijkstra had a (solid!) point
  - Structured programming raises the level of discourse
  - Leads to better code in so many ways
- Knuth also had a point
  - There are some "well-behaved" `goto`-inspired structures, which have been given names
    - `break`, `continue`, `switch`…
    - Even `try … catch`, in a sense
  - Other are discussed
    - `break` *`label`*`;` for example, which exists in Java

`goto` Considered Harmful

- There are a number of things one cannot do with `goto` statements in C++
  - They cannot skip non-vacuous initialization
  - They cannot jump in or out of functions
  - They cannot be used in `constexpr` functions http://eel.is/c++draft/dcl.constexpr#3
  - They cannot jump *into* `try` or `catch` statements http://eel.is/c++draft/except#3
    - They can be used to jump *out* of such statements, though, just like `continue` and `break`

`goto` Considered Harmful

- Am I saying « use `goto` in your code »?

`goto` Considered Harmful

- Am I saying « use `goto` in your code »?
  - Not really
  - I am saying « you're probably using `goto` or `goto`-inspired statements in your code », though
    - The point is to use higher-level abstractions as much as possible

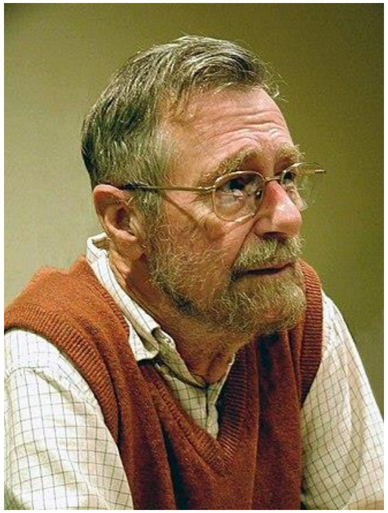
`goto` Considered Harmful

- Am I saying « use `goto` in your code »?
  - Not really
  - I am saying « you're probably using `goto` or `goto`-inspired statements in your code », though
    - The point is to use higher-level abstractions as much as possible
  - I'd also say « if you're using it to second-guess your compiler… You probably shouldn't »
    - And if you use `goto` for optimization, **measure**
    - Before **and** after
    - … and measure **regularly**. Some optimizations tend to go stale as optimizers get better at their game

Only One Exit Point per Function

Only One Exit Point per Function

- This one is also inspired from Edsger W. Dijkstra
  - Notes on structured programming, Technological University Eindhoven, The Netherlands, Department of Mathematics, T.H.-Report 70-WSK-03
  - http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF



(taken from Wikipedia)

Only One Exit Point per Function

- This one is also inspired from Edsger W. Dijkstra… but takes its roots from an earlier article from Corrado Böhm and Giuseppe Jacopin
  - Böhm, C. and Jacopin, G., Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules, Communications of the ACM, volume 9, number 5, pp. 366-371, May 1966
  - http://www.cs.unibo.it/~martini/PP/bohm-jac.pdf

Corrado Böhm
(taken from Wikipedia)

Only One Exit Point per Function

- "[…] *every Turing machine is reducible into, or in a determined sense is equivalent to, a program written in a language which admits as formation rules only composition and iteration*" (from the article)
  - https://en.wikipedia.org/wiki/Structured_program_theorem
  - Special thanks to Dan Saks, wise among the wise, for pointing me to that paper

Corrado Böhm
(taken from Wikipedia)

Only One Exit Point per Function

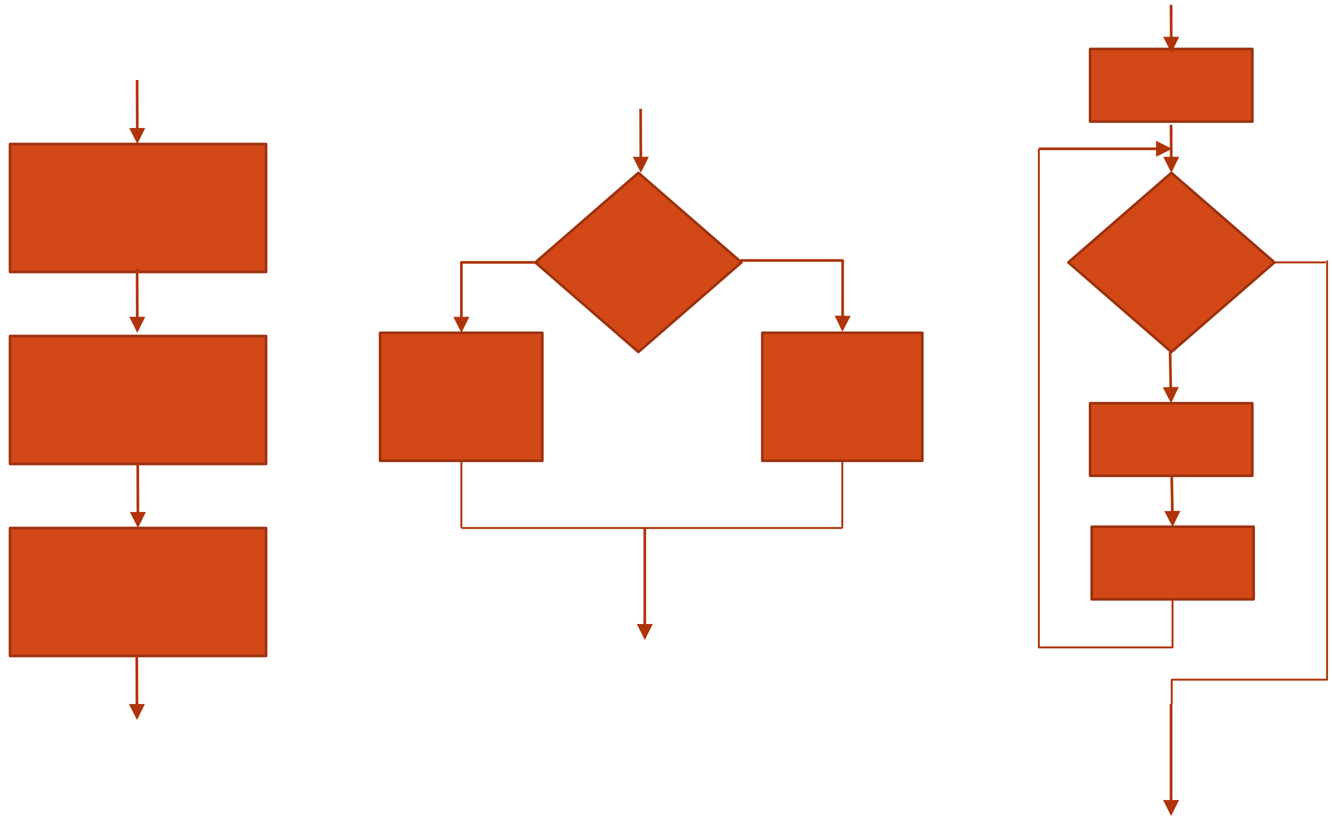
- In https://crivelloappendini.wordpress.com/2012/10/05/the-theorem-of-bohm-jacopini/, one can read "*In fact [the article] has contributed to the criticism of the injudicious use of the instructions go to is the definition of guidelines of structured programming that we have had around 1970.*"

Corrado Böhm
(taken from Wikipedia)

Only One Exit Point per Function

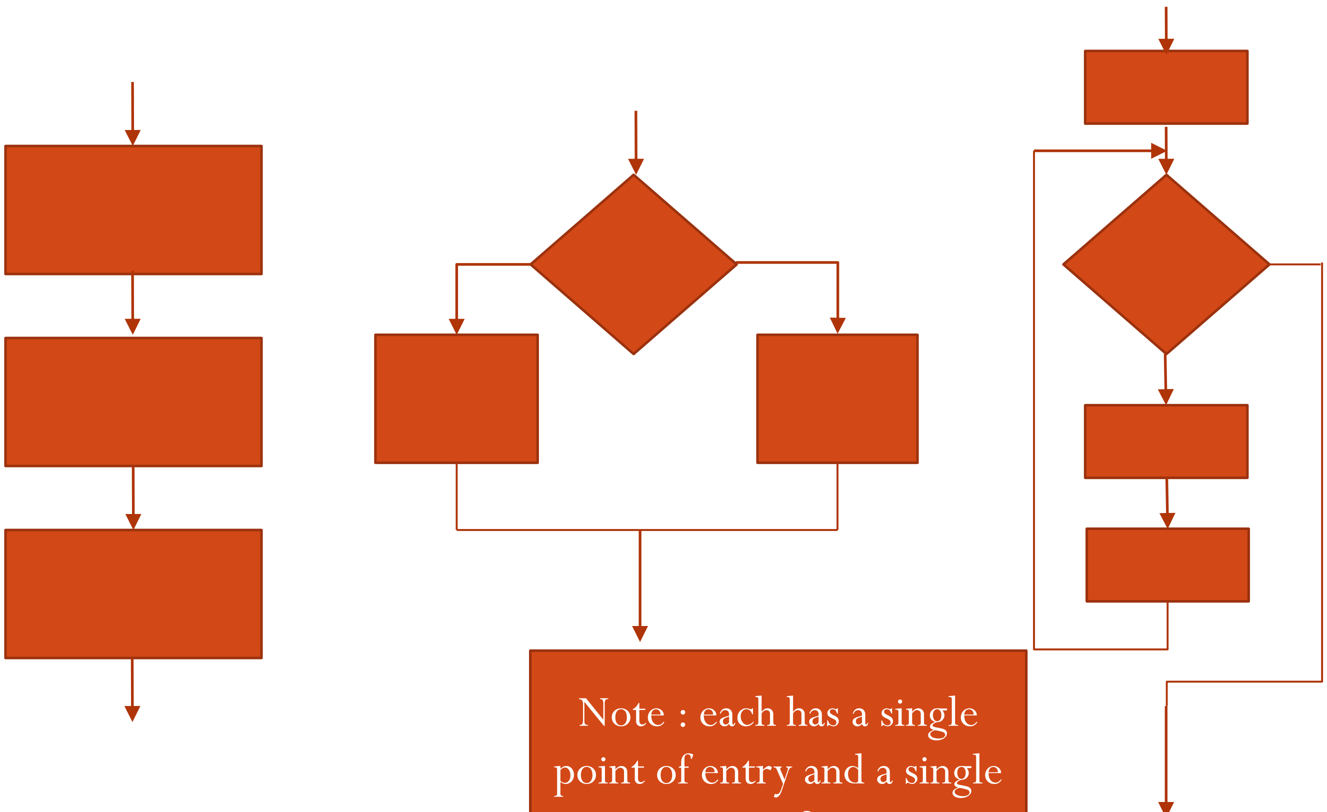
- The idea stems from a formalization of programs
  - Complex units are composed of a sequence of smaller, simpler units
  - These units are arranged in a sequence of operations
  - To achieve such a sequence, each complex operation in the sequence needs to have a single entry point and a single exit point

Only One Exit Point per Function



Basic building blocks: sequence, alternative (if-else) and repetitive (while, do…while, for)

Only One Exit Point per Function



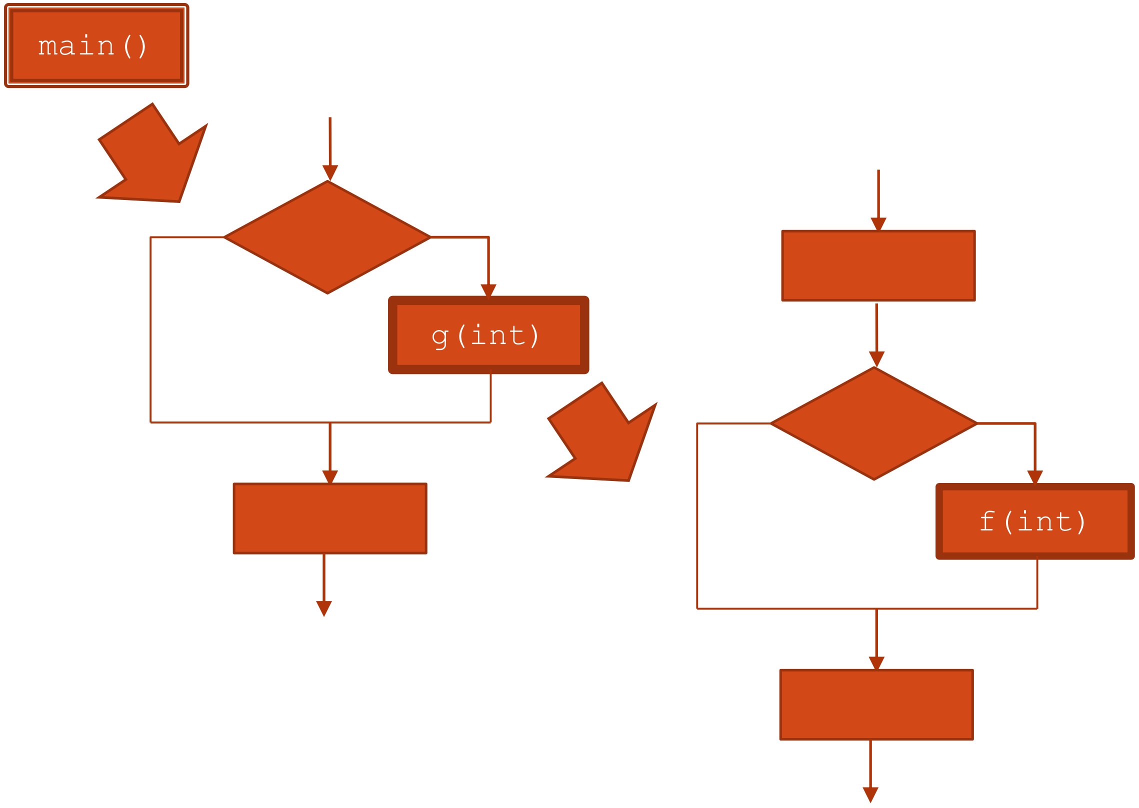
Note : each has a single point of entry and a single point of exit

Basic building blocks: sequence, a … …while, for)

Only One Exit Point per Function

- Under this perspective, functions are composite sequences

```
int f(int n) {
   return n * n;
}
int g(int n) {
   int result = 1;
   if(n < 0)
      result = f(n);
   return result;
}
int main() {
   if (int n; cin >> n)
      n = g(n);
   return n;
}
```



Only One Exit Point per Function

- There are languages where the concept of « return statement » does not exist
  - This group comprises many « Wirth languages », e.g. Pascal or Modula
  - One « sets » the return value by « assigning to the function name »
    - This does *not* conclude the function
  - When the function actually concludes, the last value written there is the actual return value

Only One Exit Point per Function

- There are languages where the concept of « return statement » does not exist
  - This group comprises many « Wirth languages », e.g. Pascal or Modula
  - One « sets » the return value by « assigning to the function name »
    - This does *not* conclude the function
  - When the function actu
    there is the actual retur

```
function factorial(n: integer): integer;
begin
  if n>1 then
    factorial:=n*factorial(n-1)
  else
    factorial:=1;
end;
```

Only One Exit Point per Function

- The single entry point, single exit point is a *reasonable* formalism
  - It helps *reason* about (structured programming) programs essentially based on their shape
  - As we have seen, however, it's not always practical to express algorithms through strict application of such a formalism
    - That does not mean one should be lax about it
    - As with many "accepted bits of wisdom", there's a lot of upsides to this one

Only One Exit Point per Function

```
// valid, but long in a way that does not
// convey useful meaning
bool is_even(int n) {
   bool result;
   if (n % 2 == 0)
      result = true;
   else
      result = false;
   return result;
}
```

Only One Exit Point per Function

```
// valid, but long in a way that does not
// convey useful meaning (not better than
// the previous one, and a bit slower on
// average)
bool is_even(int n) {
   bool result = false;
   if (n % 2 == 0)
      result = true;
   return result;
}
```

Only One Exit Point per Function

```
// valid, maybe faster, but clumsy...
// uses two exit points, but this
// provides no real upside and does
// not make code clearer in any way
bool is_even(int n) {
   if (n % 2 == 0)
      return true;
   else
      return false;
}
```

Only One Exit Point per Function

```
// lean, clean, direct. Single exit
bool is_even(int n) {
    return n % 2 == 0;
}
```

Only One Exit Point per Function

```
// early exit from loop (slow, painful)
template <class T, int N>
bool contains(T &&x, const T(&arr)[N]) {
    // also inefficient
    bool result = false;
    for(int i = 0; !result && i != N; ++i)
        if (arr[i] == x)
            result = true;
    return result;
}
```

Only One Exit Point per Function

```
// early exit from function
template <class T, int N>
bool contains(T &&x, const T(&arr)[N]) {
    for(auto & obj : arr)
        if (obj == x)
            // we have our answer
            return true;
    return false; // ok, it wasn't there
}
```

Only One Exit Point per Function

- There are many good reasons to try to achieve the single exit point principle in general
    - Among other things, this might save one from nasty bugs

Only One Exit Point per Function

```
#include <iostream>
#include <string_view>
using namespace std;
enum Color { Red, Green, Blue };
auto f(Color c) {
   switch(c) {
   case Red: return "Red"sv;
   case Green: return "Green"sv;
   case Blue: return "Blue"sv;
   }
} // probable warning here (and it's justified)
int main() {
   cout << f(Green); // what about f(10)?
}
```

Only One Exit Point per Function

```
#include <iostream>
#include <string_view>
using namespace std;
enum Color { Red, Green, Blue };
auto f(Color c) {
   auto res = "Oops"sv;
   switch(c) {
   case Red: res = "Red"sv; break;
   case Green: res = "Green"sv; break;
   case Blue: res = "Blue"sv; break;
   }
   return res;
} // Ok
int main() {
   cout << f(Green);
}
```

Only One Exit Point per Function

```
#include <iostream>
#include <string_view>
using namespace std;
enum Color { Red, Green, Blue };
auto f(Color c) {
   switch(c) {
   case Red: return "Red"sv;
   case Green: return "Green"sv;
   case Blue: return "Blue"sv;
   default: return "Oops"sv;
   }
} // Ok
int main() {
   cout << f(Green);
}
```

Only One Exit Point per Function

- One nice trick to reduce the temptation of using multiple `return` statements in a function is to write smaller functions
  - One function, one vocation
  - Sometimes, it's not quite practical to do so

Only One Exit Point per Function

```
// single vocation, many cases to cover
// (looks and feels like a flowchart)
bool is_leap_year(int year) {
   bool result;
   if (year % 400 == 0)
      result = true;
   else if (year % 100 == 0)
      result = false;
   else if (year % 4 == 0)
      result = true;
   else
      result = false;
   return result; // single exit point
}
```

Only One Exit Point per Function

```
// alternatively (similar)
bool is_leap_year(int year) {
   bool result = false;
   if (year % 400 == 0)
      result = true;
   else if (year % 100 != 0 &&
               year % 4 == 0)
      result = true;
   return result; // single exit point
}
```

Only One Exit Point per Function

```
// single vocation, many cases to cover
bool is_leap_year(int year) {
   if (year % 400 == 0)
      return true;
   else if (year % 100 == 0)
      return false;
   else if (year % 4 == 0)
      return true;
   else
      return false;
} // multiple exit points, nested
```

Only One Exit Point per Function

```
// single vocation, many cases
// to cover
bool is_leap_year(int year) {
   if (year % 400 == 0) return true;
   if (year % 100 == 0) return false;
   if (year % 4 == 0) return true;
   return false;
} // multiple exit points, flatter
  // (somewhat simpler to understand
  // and to debug)
// note : you can write it as a single
// return statement, but it's painful
// to read and understand
```

Only One Exit Point per Function

- When the return value from a function is a type with non-trivial constructors and assignment, it can be particularly useful to use multiple return statements from a function
  - `std::variant` is the posterchild for this
  - Let's examine a more complex use case

Only One Exit Point per Function

```
enum class MsgType : uint16_t { Info, Warning };
ostream &operator<<(ostream &os,
                    const MsgType &type) {
   return os << static_cast<
      underlying_type_t<MsgType>
   >(type);
}
istream &operator>>(istream &is, MsgType &type) {
   if (!is) return is;
   if (underlying_type_t<MsgType> val; is >> val)
      type = static_cast<MsgType>(val);
   return is;
}
// ...
```

Only One Exit Point per Function

```
// ...
enum class Severity : uint16_t { Low, Medium, High };
ostream &operator<<(ostream &os,
                    const Severity &severity) {
   return os << static_cast<
      underlying_type_t<Severity>
   >(severity);
}
istream &operator>>(istream &is, Severity &severity) {
   if (!is) return is;
   if (underlying_type_t<Severity> val; is >> val)
      severity = static_cast<Severity>(val);
   return is;
}
// ...
```

Only One Exit Point per Function

```
// ...
struct MsgInfo {
   string info{};
   MsgInfo() = default;
   MsgInfo(string_view info) : info{ begin(info), end(info) } {
   }
};
ostream &operator<<(ostream &os, const MsgInfo &msg_info) {
   return os << quoted(msg_info.info);
}
istream &operator>>(istream &is, MsgInfo &msg_info) {
   if (!is) return is;
   if (string info; is >> quoted(info))
      msg_info = MsgInfo{ info };
   return is;
}
// ...
```

Only One Exit Point per Function

```
// ...
struct MsgWarning {
   string warning{};
   Severity severity{ Severity::Low };
   MsgWarning() = default;
   MsgWarning(string_view warning, Severity severity)
      : warning{ begin(warning), end(warning) }, severity{ severity } {
   }
};
ostream &operator<<(ostream &os, const MsgWarning &msg_warning) {
   return os << quoted(msg_warning.warning) << ' ' << msg_warning.severity;
}
istream &operator>>(istream &is, MsgWarning &msg_warning) {
   if (!is) return is;
   if (string warning; is >> quoted(warning))
      if (Severity severity; is >> severity)
         msg_warning = { warning, severity };
   return is;
}
// ...
```

Only One Exit Point per Function

- Note that multiple exit points are typical of I/O functions
  - Particularly true of input functions
  - There's typically no reason to continue when a stream is found to be corrupted

Only One Exit Point per Function

```
// ...
template <class ... Ts>
struct Overload : Ts... {
   Overload(Ts ... ts) : Ts{ ts }... {
   }
   using Ts::operator()...;
};
template <class ... Ts>
Overload(Ts...)->Overload<Ts...>;
using Msg = variant<
   monostate, MsgInfo, MsgWarning
>;
// ...
```

Only One Exit Point per Function

```
// ...
ostream &operator<<(ostream &os, const Msg &msg) {
   return visit(Overload{
      // should never happen
      [&os](monostate) -> ostream & { return os; },
      [&os](const MsgInfo & msg_info) -> ostream& {
         return os << MsgType::Info << ' ' << msg_info;
      },
      [&os](const MsgWarning & msg_warning) -> ostream& {
         return os << MsgType::Warning << ' ' << msg_warning;
      }
   }, msg);
}
// ...
```

Only One Exit Point per Function

```
// ...
Msg consume(istream &is) {
   assert(!!is);
   Msg msg; // default ctor (costs something, no added value)
   if (MsgType type; is >> type) {
      switch (type) {
      case MsgType::Info:
         if (MsgInfo info; is >> info)
            msg = info; // replace the dummy default
         break;
      case MsgType::Warning:
         if (MsgWarning warning; is >> warning)
            msg = warning; // replace the dummy default
         break;
      }
   }
   return msg; // single exit point, but could be monostate :/
               // (requires added validation at call site)
}
// ...
```

Only One Exit Point per Function

- By imposing ourselves a single point of exit, we incur costs
  - Added complexity
  - Time to construct a dummy, empty object that would normally never be needed
  - Assign to that object with the actual objects consumed
    - This means destroying the unnecessary default object and replacing it with the object that should have been there in the first place

Only One Exit Point per Function

```
// ...
Msg consume(istream &is) {
   assert(!!is);
   if (MsgType type; is >> type) {
      switch (type) {
      case MsgType::Info:
         if (MsgInfo info; is >> info)
            return { info };
      case MsgType::Warning:
         if (MsgWarning warning; is >> warning)
            return { warning };
      }
   }
   return {}; // stream was corrupted
}
// ...
```

Only One Exit Point per Function

```
// ...
class unknown_message_type{};
Msg consume(istream &is) {
   assert(!!is);
   if (MsgType type; is >> type) {
      switch (type) {
      case MsgType::Info:
         if (MsgInfo info; is >> info)
            return { info };
      case MsgType::Warning:
         if (MsgWarning warning; is >> warning)
            return { warning };
      }
   }
   throw unknown_message_type{}; // we can get rid of monostate!
}
// ...
```

Only One Exit Point per Function

- By allowing multiple points of exit, we
  - Reduce source code complexity somewhat
  - Only construct objects that are actually needed
  - Pay the costs for a default object only when this is meaningful

Only One Exit Point per Function

- This is not a recommendation to spread functions with multiple return statements thoughtlessly
  - Writing small functions tends to reduce the temptation to resort to this approach
    - It also generally leads to better codegen!

Only One Exit Point per Function

- This is not a recommendation to spread functions with multiple return statements thoughtlessly
  - Writing small functions tends to reduce the temptation to resort to this approach
    - It also generally leads to better codegen!
  - However, having multiple exit points in a function is not inherently bad
    - Can lead to more efficient code
    - Can save unnecessary construction of dummy objects
    - Can make code easier to understand and debug

Make All Data Members Private

Make All Data Members Private

- This is typically seen as good practice
  - Enforces encapsulation by putting the object in control of its internal states

Make All Data Members Private

```
class invalid_name{};
class invalid_age{};
class Person {
   string name_;
   short age_;
   // business logic (used to enforce class invariants)
   static constexpr auto validate_name(string_view name) {
      return name.empty()? throw invalid_name{} : name;
   }
   static constexpr auto validate_age(short age) {
      return age < 0? throw invalid_age{} : age;
   }
public:
   Person(string_view name, short age)
      : name_{ validate_name(name) }, age_{validate_age(age) } {
   }
   string name() const { return name_; }
   short age() const { return age_; }
};
```

Make All Data Members Private

```
class invalid_name{};
class invalid_age{};
class Person {
public:
   string name; // publicly accessible (no way to enforce
   short age;   // class invariants without discipline...)
private:
   // business logic (used to enforce class invariants)
   static constexpr auto validate_name(string_view name) {
      return name.empty()? throw invalid_name{} : name;
   }
   static constexpr auto validate_age(short age) {
      return age < 0? throw invalid_age{} : age;
   }
public:
   // false sense of security
   Person(string_view name, short age)
      : name{ validate_name(name) }, age{validate_age(age) } {
   }
};
```

Make All Data Members Private

```
class invalid_name{};
class invalid_age{};
struct Person {
   string name; // publicly accessible (no way to enforce
   short age;   // class invariants without discipline...)
private:
   // business logic (used to enforce class invariants)
   static constexpr auto validate_name(string_view name) {
      return name.empty()? throw invalid_name{} : name;
   }
   static constexpr auto validate_age(short age) {
      return age < 0? throw invalid_age{} : age;
   }
public:
   // false sense of security
   Person(string_view name, short age)
      : name{ validate_name(name) }, age{validate_age(age) } {
   }
};
```

Make All Data Members Private

- This is typically seen as good practice
  - Enforces encapsulation by putting the object in control of its internal states
  - However, not all types have invariants to enforce

Make All Data Members Private

```
class Point {
   int x_{}, y_{}, z_{};
public:
   Point() = default;
   // note : all possible states are
   // acceptable
   constexpr Point(int x, int y, int z)
      : x_{x}, y_{y}, z_{z} {
   }
   int x() const noexcept { return x_; }
   int y() const noexcept { return y_; }
   int z() const noexcept { return z_; }
   // ...
};
```

Make All Data Members Private

```
struct Point {
    int x{}, y{}, z{};
    Point() = default;
    // note : all possible states are
    // acceptable (no invariants)
    constexpr Point(int x, int y, int z)
        : x{ x }, y{ y }, z{ z } {
    }
    // ...
};
```

Make All Data Members Private

- This is typically seen as good practice
  - Enforces encapsulation by putting the object in control of its internal states
  - However, not all types have invariants to enforce
- A case could be made for "thinking ahead", and making it easier to add invariants later
  - This comes at the cost of added complexity
    - Very *non*-YAGNI (*You Ain't Gonna Need It*)
    - C# or Delphi-like properties could help somewhat here, but C++ does not currently support such a feature

Make All Data Members Private

- There's a simple trick here: think the design through
  - Try to write simpler classes
  - Document invariants
  - Make data members private in general
- Public data members work well for classes that…
  - Have no invariants to enforce
  - Are mutable
  - Are `final`
    - Inheriting from such a class should probably only be done through private inheritance
    - Otherwise, prefer composition over inheritance

Make All Data Members Private

- Note that this:

```
struct Error {
   const string message;
   Error(string_view msg) : message{ begin(msg), end(msg) }{ }
};
```

- … is not a semantic equivalent for that:

```
class Error {
   string msg;
public:
   Error(string_view msg) : msg{ begin(msg), end(msg) } { }
   string message() const {
      return msg;
   }
};
```

- … as there are consequences that might surprise one going from one to the other

Initialize All Variables on Definition

Initialize All Variables on Definition

- Ergo, instead of this:

```
int n;
float f;
string s;
vector<X> v;
```

- … go for:

```
int n = 0;
float f = 0.0f;
string s = "";
vector<X> v = vector<X>();
```

Initialize All Variables on Definition

- Ergo, instead of this:

```
int n;
float f;
string s;
vector<X> v;
```

- … or go for:

```
int n = {};
float f = {};
string s = {};
vector<X> v = {};
```

Initialize All Variables on Definition

- Ergo, instead of this:

```
int n;
float f;
string s;
vector<X> v;
```

- … or go for:

```
int n {};
float f {};
string s {};
vector<X> v {};
```

Initialize All Variables on Definition

- Ergo, instead of this:

```
int n;
float f;
string s;
vector<X> v;
```

- … but *do not* go for this:

```
int n(); // do you see it?
float f();
string s();
vector<X> v();
```

Initialize All Variables on Definition

- Ergo, instead of this:

```
int n;
float f;
string s;
vector<X> v;
```

- … or go for:

```
int n {};
float f {};
string s; // Ok, implicit default ctor
vector<X> v; // likewise
```

- … unless writing generic code, of course

Initialize All Variables on Definition

- One of the key tenets of C++ is "You do not pay for what you do not use"
  - We strive hard to keep it that way
    - There are still some costs inherent to contemporary C++, but we're working on it
  - Now, is this…

```
int n = 0;
```

  - … really more costly than this?

```
int n;
```

Initialize All Variables on Definition

- One of the key tenets of C++ is "You do not pay for what you do not use"
  - We strive hard to keep it that way
    - There are still some costs inherent to contemporary C++, but we're working on it
  - Now, is this…

```
int n = 0;
```

  - … really more costly than this?

```
int n;
```

  - Actually, it sometimes is
    - …and C++ has users that cannot tolerate such costs

Initialize All Variables on Definition

- C++ distinguishes declaration from definition

```
void f(); // declaration
void f(); // Ok, declaration can be repeated
struct X {
   friend void f(); // declaration, still Ok
};
extern X x; // declaration
X x; // definition (object is created)
void f() { } // definition
// not Ok (would be an ODR violation)
// X x;
// not Ok (would be an ODR violation)
// void f() {} // not Ok (likewise)
```

Initialize All Variables on Definition

- Some languages "enforce" a rule about object initialization
  - In C#, this does not compile:

```
// zeroed but conceptually uninitialized
int n;
// error, n not initialized (the equivalent
// code would compile but be UB in C++)
Console.WriteLine(n);
```

  - However, in C#, this does compile:

```
// zeroed, technically and conceptually
int [] ns = new int[10];
// compiles, prints 0
Console.WriteLine(ns[3]);
```

Initialize All Variables on Definition

- There's really no point to doing this…

```
int n = 0; // ...why?
if (cin >> n) // ...we're writing to n here
    f(n); // only reached if input succeeded
```

- … instead of doing this:

```
int n; // unitialized
if (cin >> n) // ... might be initialized
    f(n); // ... only if it is initialized
```

- … or even better (if the scope of `n` is the `if` statement), this:

```
if (int n; cin >> n)
    f(n);
```

Initialize All Variables on Definition

- The point of this principle is to avoid leaving uninitialized variables lying around in the code
  - … for good reason!
- Many languages have traditionally imposed the definition of variables at the beginning of blocks
- However, languages that support object orientation lead us to think about initialization costs
  - Constructors are often not free
- Variables that are initialized somewhere, but used far away are actually bad code smells!
  - … and a sign that the function is too big!

Initialize All Variables on Definition

```
class Person {
   // ...
public:
   Person(string_view name, short age);
   // ...
   friend istream& operator>>(istream &is, Person &p) {
      string name; // bad, might not be
      short age;   // needed in practice
      if (!is) return is;
      if (is >> name >> age)
         p = { name, age };
      return is;
   }
};
```

Initialize All Variables on Definition

```
class Person {
   // ...
public:
   Person(string_view name, short age);
   // ...
   friend istream& operator>>(istream &is, Person &p) {
      if (!is) return is;
      string name; // better
      short age;   // a bit early (but trivial ctor)
      if (is >> name >> age)
         p = { name, age };
      return is;
   }
};
```

Initialize All Variables on Definition

```
class Person {
   // ...
public:
   Person(string_view name, short age);
   // ...
   friend istream& operator>>(istream &is, Person &p) {
      if (!is) return is;
      short age;   // a bit early (but trivial ctor)
      if (string name; is >> name >> age)
         p = { name, age };
      // note : age still exists here... why?
      return is;
   }
};
```

Initialize All Variables on Definition

```
class Person {
   // ...
public:
   Person(string_view name, short age);
   // ...
   friend istream& operator>>(istream &is, Person &p) {
      if (!is) return is;
      // better still (scopes are shorter)
      if (string name; is >> name)
         if(short age; is >> age)
            p = { name, age };
      return is;
   }
};
```

Initialize All Variables on Definition

- When facing a two-step initialization, consider encapsulating it in a factory function
  - This avoids leaking the uninitialized objet into client code

Initialize All Variables on Definition

```
class TwoStepThing {
   // ...
public:
   void Init() { // must be called only once *this is fully constructed
      // ...
   }
protected:
   TwoStepThing() = default;
   friend TwoStepThing create_two_step();
   // ...
};
TwoStepThing create_two_step() {
   TwoStepThing thing; // probably uninitialized
   thing.Init(); // now probably Ok
   // ...
   return thing; // return fully initialized object
}
int main() {
   auto thing = create_two_step();
}
```

Initialize All Variables on Definition

- This "don't declare variables before you need them" rule is also true of languages such as Java or C# where objects are only accessed indirectly
  - Given this function:

```
static X makeX(/* args */) { /* ... */ }
```

  - Something like this Java code:

```
static void useX() {
    X x; // uninitialized (null by default)
    // ... do things; x remains uninitialized (bad!)...
    x = makeX( /* args */ );
    // ...
}
```

  - … would be much better as

```
static void useX() {
    // ... do things ...
    X x = makeX( /* args */ ); // declared when needed
    // ...
}
```

Initialize All Variables on Definition

- The usual tricks apply in this case
  - Think before you code
  - Write small functions
  - Declare objects only when… and where needed
- Keeping objects close to point of use tends to reduce the tendency to use superfluous initialization
  - So does accepting some unusual control flow constructs
    - Occasionally
  - So does accepting multiple points of exit in a function
    - When justified

So…

So…

- Avoid `goto` unless you can justify it
  - There are probably `goto`-like things in your code base, though
    - It's not *that* dirty
    - … but it is a code smell
- Keep your code at a higher level of abstraction

So...

- Multiple returns from a function are not *that* bad
  - Keep your functions small, their vocation clear
  - Make sure these multiple exits have a (documented) reason for being
    - Do things for a reason, not just for laziness

So...

- No, you don’t need to initialize all variables at their point of definition
  - Sometimes wasteful
  - Define them when needed only
    - That way, if they are initialized-after-definition, they don’t stay uninitialized for long
  - If you need two-step initialization, encapsulate it in a factory

So…

- No, you don't need to make all member variables private
  - It eases refactoring, though
  - … but it's really useful when you have invariants to enforce
    - No, protected member variables are not better than public ones
      - Or if they are, it's just barely

Questions?

Some Other Myths (if there's time)

Some Other Myths

- There's a bunch of speed-related myths, e.g.:
  - Vector slower than dynamically allocated array
  - Objects slower than functions
  - Smart pointers slower than raw pointers

Some Other Myths

- Vector slower than dynamically allocated array?
- It depends
  - Construction (if done properly)
    - http://quick-bench.com/AguBHmz3EJatWDP8j8x9pWMvHiI

Some Other Myths

- Vector slower than dynamically allocated array?
- It depends
  - Construction (if done properly)
    - http://quick-bench.com/AguBHmz3EJatWDP8j8x9pWMvHiI
    - `vector` distinguishes allocation and initialization
    - Can be made to do the minimum effort easily
    - … and is safer!

Some Other Myths

```
{
   // calls T::T() n times (conceptually, size
   // and capacity of the array are both n)
   T *p = new T[n];
   // ... hope no exception gets thrown here!
   delete [] p;
}
{
   // calls T::T() n times (v.size()==n ; v.capacity()==n)
   vector<T> v(n);
   // ...
} // resources freed no matter what happens
{
   vector<T> v; // no call to T::T() (much faster if non-vacuous!)
   v.reserve(n); // v.size() == 0; v.capacity() == n
   // ... add T elements through push_back()/emplace_back() when ready
}
```

Some Other Myths

- Vector slower than dynamically allocated array?
- It depends
  - Construction (if done properly)
    - http://quick-bench.com/AguBHmz3EJatWDP8j8x9pWMvHiI
  - Construction and initialization (if done properly)
    - http://quick-bench.com/ToZLo4zagY2Z-U-vER2kVik5hlg

Some Other Myths

```
{
   // calls T::T() n times (wasteful!)
   T *p = new T[n];
   fill(p, p + n, value); // calls T::operator= n times!
   // ...
   delete [] p;
}
{
   // calls T::T() n times (v.size()==n ; v.capacity()==n)
   vector<T> v(n);
   fill(begin(v), end(v), value); // calls T::operator= n times!
   // ...
}
{
   vector<T> v; // no call to T::T() (much faster if non-vacuous!)
   v.reserve(n); // v.size() == 0; v.capacity() == n
   fill_n(back_inserter(v), n, value); // calls T::T(const T&) n times
}
```

Some Other Myths

- To do the equivalent of this code by hand, with an array…

```
vector<T> v;
v.reserve(n);
fill_n(back_inserter(v), n, value);
// ...
```

Some Other Myths

- … one has to get one's hands slightly dirty:

```
T *p = reinterpret_cast<T*>(
   new /*(align_val_t{ alignof(T) })*/
      char[n * sizeof(T)]
);
int i = 0;
try {
   for(; i != n; ++i)
      new (static_cast<void*>(p + i)) T{value};
} catch(...) {
   for(int j = i; --j >= 0; )
      (p + i)->~T();
   delete [] reinterpret_cast<char*>(p);
   throw;
}
```

Some Other Myths

- … a bit less painful with algorithms:

```
T *p = reinterpret_cast<T*>(
   new /*(align_val_t{ alignof(T) })*/
      char[n * sizeof(T)]
);
try {
   uninitialized_fill(p, p + n, value);
} catch(...) {
   delete [] reinterpret_cast<char*>(p);
   throw;
}
```

Some Other Myths

- … a bit less painful with algorithms and a smart pointer:

```
unique_ptr<T[], function<void(T*)>> p{
   reinterpret_cast<T *>(
      new /*(align_val_t{ alignof(T) })*/
         char[n * sizeof(T)]
   ),
   [](T * p) {
      delete[]reinterpret_cast<char *>(p);
   }
};
uninitialized_fill(&p[0], &p[n], value);
```

Some Other Myths

- … a bit less painful with algorithms and a smart pointer:

```
unique_ptr<T[], function<void(T*)>> p{
   reinterpret_cast<T *>(
      new /*(align_val_t{ alignof(T) })*/
         char[n * sizeof(T)]
   ),
   [](T * p) {
      delete[]reinterpret_cast<char *>(p);
   }
};
uninitialized_fill(&p[0], &p[n], value);
```

- But really: use `vector`, please!

Some Other Myths

- Vector slower than dynamically allocated array?
- It depends
  - Construction (if done properly)
    - http://quick-bench.com/AguBHmz3EJatWDP8j8x9pWMvHiI
  - Construction and initialization (if done properly)
    - http://quick-bench.com/ToZLo4zagY2Z-U-vER2kVik5hlg
  - It's always better to measure, but in general you can trust your compiler vendors and your library writers
    - Lots of *very* smart people there!

Some Other Myths

- Objects slower than functions?

Some Other Myths

- Objects slower than functions?
  - It depends
    - Applying equivalent operations many times: http://quick-bench.com/l-3FPjpStsMfFhOu5EzMhw8ghXs
  - Again: it's always better to measure, but you in general, can trust your compiler vendors and your library writers
    - Lots of *very* smart people there!
    - Compilers are very, very good
    - Functors are very, very useful
    - Lambdas are just awesome

Some Other Myths

- Smart pointers slower than raw pointers?

Some Other Myths

- Smart pointers slower than raw pointers?
  - It depends

```
int main() {
    int *p = new int{ 3 };
    int n = *p;
    delete p;
    return n;
}
```

    - -O0 with gcc: https://godbolt.org/z/xHc9Ei
    - -O2 with gcc: https://godbolt.org/z/CtQOcQ
    - -O0 with clang: https://godbolt.org/z/xQBZZy
    - -O2 with clang: https://godbolt.org/z/XWuSQK (no joke!)

Some Other Myths

- Smart pointers slower than raw pointers?
  - It depends

```
int main() {
   unique_ptr<int> p { new int{ 3 } };
   return *p;
}
```

    - -O0 with gcc: https://godbolt.org/z/I9JBhb
    - -O2 with gcc: https://godbolt.org/z/QCW37v
    - -O0 with clang: https://godbolt.org/z/ih2LGa
    - -O2 with clang: https://godbolt.org/z/m3YuQN (no joke!)

Some Other Myths

- Smart pointers slower than raw pointers?
  - It depends
  - `unique_ptr` costs essentially nothing in time or space (apart from a few, niche details) with respect to raw pointers
    - One additional operation on movement
    - Space overhead to store the custom delete when it's a function pointer

Some Other Myths

- Smart pointers slower than raw pointers?
  - It depends
  - `unique_ptr` costs essentially nothing in time or space (apart from a few, niche details) with respect to raw pointers
    - One additional operation on movement
    - Space overhead to store the custom delete when it's a function pointer
  - `shared_ptr` costs a lot more, but has a very specific niche
    - You should not be using it much
    - If you use it, on the other hand, it's probably much better than anything you could hand-write (it's very tricky code)

Some Other Myths

- Smart pointers slower than raw pointers?
  - It depends
  - Again and again: it's always better to measure, but you in general, can trust your compiler vendors and your library writers
    - Lots of *very* smart people there!

So…

So...

- Use vector
- Use functors
- Use lambdas
    - … measure, but give your standard library tools and your compiler a chance

- … and enjoy C++!

Questions?