Basic C++

Overloading and OOP

Dr. Porkoláb Zoltán Károly

gsd@inf.elte.hu

http://gsd.web.elte.hu

Be careful with optimization!

```
int main()
{
    const int ci = 10;
    int *ip = const_cast<int*>(&ci);
    ++*ip;
    cout << ci << " " << *ip << endl;
    return 0;
}
```

```
$ ./a.out
```

Be careful with optimization!

```
int main()
{
    const int ci = 10;
    int *ip = const_cast<int*>(&ci); // undefined behavior
    ++*ip;
    cout << ci << " " << *ip << endl;
    return 0;
}
```

```
$ ./a.out
10 11
```

Be careful with optimization!

```
int main()
{
    volatile const int ci = 10;
    int *ip = const_cast<int*>(&ci); // undefined behavior
    ++*ip;
    cout << ci << " " << *ip << endl;
    return 0;
}
```

```
$ ./a.out
11 11
```

Functions

- Fundamental building blocks
  - Should be declared before first use
  - Declaration is local or global
  - Definition is always global (no nested functions)
- Function associates
  - Function name
  - Block: sequence of statements
  - Return type (or void)
  - Parameters (can be defaulted)

Functions

```
bool isodd( int n); // functions should be declared before first use

int f()
{
    for ( auto arg : { 1, 3, 42, -5, 8} )    // arg is int
    {
        std::cout << isodd(arg) << ' '; // function call: arg copied to param
    }
}

bool isodd( int n)  // arg is copied to n for every call
{
    return n % 2;
}
```

Functions

```
bool isodd( int n); // formal parameters can be ignored

int f()
{
    for ( auto arg : { 1, 3, 42, -5, 8} )    // arg is int
    {
        std::cout << isodd(arg) << ' '; // function call: arg copied to param
    }
}

bool isodd( int n)  // arg is copied to n for every call
{
    return n % 2;
}
```

Functions

```
bool isodd( int ); // functions should be declared before first use

int f()
{
    for ( auto arg : { 1.5, 3.14, 42.0, -5.5, 8.8} )  // arg is double
    {
        std::cout << isodd(arg) << ' '; // function call: arg copied to param
    }
}

bool isodd( int n)  // arg is converted to int then copied to n
{
    return n % 2;
}
```

Functions

```
bool isodd( int ); // functions should be declared before first use

int f()
{
    for ( auto arg : { "1", "3", "42", "-5", "8"} )  // arg is const char *
    {
        std::cout << isodd(arg) << ' '; // compile error
    }                                   // arg does not convertible to int
}

bool isodd( int n)
{
    return n % 2;
}
```

Function declaration

```
int f( int i, double d, std::string s);  // function declaration
int f( int  , double  , std::string  );  // same as above

int g();         // function with no parameter, different from C
int g(void);     // C style declaration, same as above

void h( int, double, std::string);  // function with no return type

int printf( const char *format, ...);  // one or more parameters
int fprintf( FILE *fp, const char *format ...);  // two or more parameters
int fdots(...); // zero or more parameters, we do know nothing about the pars

template <typename ...Args>    // variadic function template
void tf( Args ...pars);

date_t make_date( int year, int month = 1, int day = 1); // default arguments
```

Function declaration

```
int f( int i, double d, std::string s);  // function declaration
int f( int  , double  , std::string  );  // same as above

int g();         // function with no parameter, different from C
int g(void);     // C style declaration, same as above

void h( int, double, std::string);  // function with no return type

int printf( const char *format, ...);  // one or more parameters
int fprintf( FILE *fp, const char *format ...);  // two or more parameters
int fdots(...); // zero or more parameters, we do know nothing about the pars

template <typename ...Args>    // variadic function template
void tf( Args ...pars);

date_t make_date( int year, int month = 1, int day = 1); // default arguments
date_t make_date( int year, int = 1, int = 1); // default arguments
```

Default arguments

- Default arguments belong to the caller environment
- Should be continuous from right-to-left
- Can be different in independent scopes

```
void f()
{
  date_t make_date( int year, int month = 1, int day = 1);  // default args

  date_t d1 = make_date( 2023, 8, 14);
  date_t d1 = make_date( 2023, 8);  // make_date( 2023, 8, 1)
  date_t d1 = make_date( 2023);     // make_date( 2023, 1, 1)
}
void g()
{
  date_t make_date( int year, int month = 12, int day = 25);
  date_t xmas = make_date( 2023);   // make_date( 2023, 12, 25)
}
```

Function definition

```
double fahr2cels( double f)
{
    return 5./9.*(f-32);
}

auto cels2fahr( double c)  // return type deduction
{
    return c*9./5.+32;
}

void f( double par1, int )    //  unused parameters may not named
{
    return fahr2cels(par1);
}

int fact( int f)  // recursive function
{
    if ( 1 == n )
        return 1;
    else
        return n*fact(n-1);    // tail recursion, can be optimized
}
```

Parameter passing by value

```
#include <iostream>

void increment(int par)    // creates a copy of actual argument of i in main
{
  ++par;   // increments local copy
  std::cout << "i in increment() " << par << std::endl;
}

int main()
{
  int i = 0;
  increment(i);  // copies 0 to par
  increment(i);  // copies 0 to par
  std::cout << "i in main() = " << i << std::endl;
  return 0;
}

$ g++ -ansi -pedantic -Wall -W par.cpp
$ ./a.out
i in increment() = 1
i in increment() = 1
i in main() = 0
```

Parameter passing by “address”

```
#include <iostream>

void increment(int *par)   // par points to the memory area of i in main
{
  ++*par;  // increments i in main
  std::cout << "i in increment() " << *par << std::endl;
}

int main()
{
  int i = 0;
  increment(&i);  // copies the address of i to par
  increment(&i);  // copies the address of i to par
  std::cout << "i in main() = " << i << std::endl;
  return 0;
}

$ g++ -ansi -pedantic -Wall -W par.cpp
$ ./a.out
i in increment() = 1
i in increment() = 2
i in main() = 2
```

Parameter passing by reference

```
#include <iostream>

void increment(int &par)   // par refers to the memory area of i in main
{
  ++par;   // increments i in main
  std::cout << "i in increment() " << par << std::endl;
}

int main()
{
  int i = 0;
  increment(i);  // initialize par (reference) to the memory area of i
  increment(i);  // initialize par (reference) to the memory area of i
  std::cout << "i in main() = " << i << std::endl;
  return 0;
}

$ g++ -ansi -pedantic -Wall -W par.cpp
$ ./a.out
i in increment() = 1
i in increment() = 2
i in main() = 2
```

Parameter passing

- Parameter passing in C++ follows initialization semantics
  - Value initialization copies the object
  - Reference initialization just set up an alias

```
void f1( int x, int y) { ... }
void f2( int &xr, int &yr) { ... }

int i = 5, j = 6;

f1( i, j);   int x = i;  // creates local x and copies i to x
             int y = j;  // creates local y and copies j to y

f2( i, j);   int &xr = i; // binds xr name to i outside
             int &yr = j; // binds yr name to j outside
```

Swap before move semantics

```
void swap( int &x, int &y)
{
  int tmp = x;
  x = y;
  y = tmp;
}

int main()
{
  int i = 5;
  int j = 6;

  swap( i, j);
  assert(i==6 && j==5);
}
```

Swap with move semantics

```
void swap( int &x, int &y)
{
  int tmp = std::move(x);
  x = std::move(y);
  y = std::move(tmp);
}

int main()
{
  int i = 5;
  int j = 6;

  swap( i, j);
  assert(i==6 && j==5);
}
```

Function overloading

- Overload resolution happens when function name is not unique
- Overload resolution process
  - Building candidate function set
  - Reduction to viable functions
  - Find the best viable function (if exists)

Function overloading

- **No overloading in C**, all function names must be unique

```
#include <stdlib.h>
int           abs(int n);
long          labs(long n);
long long     llabs(long long n);    // C99
intmax_t      imaxabs( intmax_t n); // C99

#include <math.h>
double        fabs(double n);
float         fabsf(float n);        // C99
long double   fabsl(long double n); // C99
```

- Other languages: C++, Java, C#, Rust, etc. allows if **signature is different**

```
#include <cmath>   // C++

int         abs(int n);
long        abs(long n);
long long   abs(long long n);
double      abs(double n);
// ...
```

Overload resolution step 1

- Candidate functions
  - Finding the callable entities which match to the call expression
- Viable functions
  - The same number of parameters as in the call expression
  - Less parameters, but there is ellipsis parameter (...)
  - Less parameters, but the rest has/have default arguments
  - Constraints are satisfied (C++20)
  - Every argument is convertible to the corresponding parameter

Overload resolution step 2

- Best viable function
  - Ranking the conversion from actual argument to formal one
  - If there is a parameter where the conversion is better and the all other parameters are not worse
- Ranking of conversion (best to worse)
  - No or trivial conversions
  - Arithmetic promotions
  - Static cast
  - User defined conversions

Overload resolution

```
void f(const int *, short);
void f(int *, int);

int main()
{
  int   i = 0;
  short s = 0;
  long  l = 10L;

  f( &i, l);


  f( &i, 'c');


  f( &i, s);

}
```

Overload resolution

```
void f(const int *, short);
void f(int *, int);

int main()
{
  int   i = 0;
  short s = 0;
  long  l = 10L;

  f( &i, l);   // (1) &i->int* > &i->const int* (2) l->int = l->short


  f( &i, 'c'); // (1) &i->int* > &i->const int* (2) 'c'->int > 'c'->short


  f( &i, s);   // (1) &i->int* > &i->const int* (2) s->int < s->short

}
```

Overload resolution

```
void f(const int *, short);
void f(int *, int);

int main()
{
  int   i = 0;
  short s = 0;
  long  l = 10L;

  f( &i, l);   // (1) &i->int* > &i->const int* (2) l->int = l->short
               // calls f(int *, int)

  f( &i, 'c'); // (1) &i->int* > &i->const int* (2) 'c'->int > 'c'->short
               // calls f(int *, int)

  f( &i, s);   // (1) &i->int* > &i->const int* (2) s->int < s->short
               // compiler error
}
```

Overloading on const

```
template <typename T, ... >
class vector
{
public:
    T&       operator[](size_t i);
    const T& operator[](size_t i) const;
    // ...
};
int main()
{
          std::vector<int>   iv;
    const std::vector<int>  civ;
    // ...
    iv[i] = 42;         // non-const
    int i = iv[5];
    int j = civ[5]      // const
    // ...
}
```

Overloading on const

```
template <typename It, typename T>
It find( It begin, It end, const T& t)
{
    while (begin != end) {
        if ( *begin == t )
            return begin;
        ++begin;
    }
    return end;
}

const char t[] = { 1, 2, 3, 4, 5 };
const char *p = std::find( t, t+sizeof(t), 3)
if ( p != t+sizeof(t) )
{
    std::cout << *p; // ok to read
    // syntax error: *p = 6;
}
const std::vector<int>  v(t, t+sizeof(t));
std::vector<int>::const_iterator i = std::find( v.begin(), v.end(), 3);
if ( v.end() != i )
{
     std::cout << *i; // ok to read
    // syntax error: *i = 6;
}
```

Overloading on const

```
//  C++11


std::vector<int>  v1(4,5);
auto i = std::find( v1.begin(), v1.end(), 3);
// i is vector<int>::iterator

const std::vector<int>  v2(4,5);
auto j = std::find( v2.begin(), v2.end(), 3);
// j is vector<int>::const_iterator

auto k = std::find( v1.cbegin(), v1.cend(), 3);
// k is vector<int>::const_iterator
```

Overloading on const

```
//  C++11


std::vector<int>  v1(4,5);
auto i = std::find( std::begin(v1), std::end(v1), 3);
// i is vector::iterator

const std::vector<int>  v2(4,5);
auto j = std::find( std::begin(v2), std::end(v2), 3);
// j is vector::const_iterator

auto k = std::find( std::cbegin(v1), std::cend(v1), 3);
// k is vector::const_iterator
```

Overloading on left-right value

```
//  C++11
template <typename T, ... >
class vector
{
public:
    void      push_back(const T& t);
    void      push_back(T&& t);
    // ...
};
int main()
{
    std::vector<int>                iv;
    std::vector<int>                f();
    std::vector<std::vector<int>>   vv;

    vv.push_back(iv);  // push_back(const T& t);
    vv.push_back(f()); // push_back(T&& t);
}
```

Overloading on left-right value

```
//  C++20
template <typename T, ... >
class vector
{
public:
    constexpr void push_back(const T& t);
    constexpr void push_back(T&& t);
    // ...
};
int main()
{
    std::vector<int>                iv;
    std::vector<int>                f();
    std::vector<std::vector<int>>   vv;

    vv.push_back(iv);  // push_back(const T& t);
    vv.push_back(f()); // push_back(T&& t);
}
```

Overloading on this

```
T &       value(optional<T> &       par);
T &&      value(optional<T> &&      par);
T const&  value(optional<T> const&  par);
T const&& value(optional<T> const&& par);
```

But in object-oriented programming, sometimes we want to overload on the **this** parameter too.

```
template <typename T>
class optional
{
  // ...
  T&        value() &;
  T&&       value() &&;
  T const&  value() const&;
  T const&& value() const&&;
};
```

Overloading on this

```
template <typename T>
class optional {
  constexpr T& value() & {
    if (has_value()) {
      return this->m_value;
    }
    throw bad_optional_access();
  }
  constexpr T const& value() const& {
    if (has_value()) {
      return this->m_value;
    }
    throw bad_optional_access();
  }
  constexpr T&& value() && {
    if (has_value()) {
      return std::move(this->m_value);
    }
    throw bad_optional_access();
  }
  constexpr T const&& value() const&& {
    if (has_value()) {
      return std::move(this->m_value);
    }
    throw bad_optional_access();
  }
  // ...
};  // example from https://devblogs.microsoft.com/cppblog/cpp23-deducing-this/
```

Deducing this (C++23)

```
template <typename T>
class optional {
  constexpr T& value() & {
    if (has_value()) {
      return this->m_value;
    }
    throw bad_optional_access();
  }
  constexpr T const& value() const& {
    if (has_value()) {
      return this->m_value;
    }
    throw bad_optional_access();
  }
  constexpr T&& value() && {
    if (has_value()) {
      return std::move(this->m_value);
    }
    throw bad_optional_access();
  }
  constexpr T const&& value() const&& {
    if (has_value()) {
      return std::move(this->m_value);
    }
    throw bad_optional_access();
  }
  // ...
};  // example from https://devblogs.microsoft.com/cppblog/cpp23-deducing-this/
```

```
template <typename T>
struct optional {
  // One version of value which works for everything
  template <class Self>
  constexpr auto&& value(this Self&& self) {
    if (self.has_value()) {
        return std::forward<Self>(self).m_value;
    }
    throw bad_optional_access();
  }
```

How to define operators?

- a + b, a – b, a == b, ...

  a.**operator**+(b)

  operator+(a,b)

- a = b, a[b], a(b1,b2,...), a-> only member

  a.**operator**= (b)

  a.**operator**[ ] (b)

  a.**operator**( ) (b1, b2, …)

  a.**operator**->()

Operators

```
#include <iostream>
#include <string>

int main()
{
    int i = 42;
    std::cout << i << '\n';  // std::cout.operator<<(i).operator<<('\n');
}
```

Operators

```
#include <iostream>
#include <string>

int main()
{
    int i = 42;
    std::cout << i << '\n';  // std::cout.operator<<(i).operator<<('\n');
}
```

```
int main()
{
    std::string h = "Hello ";
    std::string w = "world\n";
    std::cout << h << w;     // std::operator<<(operator<<(cout,w),s);
}
```

Operators

```
#include <iostream>

struct complex_t
{
    double re;
    double im;
};

std::ostream& operator<< ( std::ostream& os, const complex_t &c)
{
    os << re << '+' << im << 'i';
    return os;    // important for chaining output operations
}

int main()
{
    complex_t c{1, 3.14};
    std::cout << c << '\n'; // std::operator<<(operator<<(cout,c),'\n');
}
```

Operators

```
#include <iostream>
#include <fstream>

int main()
{
    complex_t c{1, 3.14};
    std::ofstream outfile{"out.txt};

    if ( outfile )
    {
        outfile << c << '\n';
    }
}
```

Operators

```
#ifndef COMPLEX_T
#define COMPLEX_T

#include <iostream>

struct complex_t
{
    double re;
    double im;
    bool operator==( const complex_t& c) const  // similarly != < etc.
    {
        return re == c.re && im == c.im;
    }
};

#endif // COMPLEX_T

int main()
{
    complex_t c1{1, 3.14}, c2{3.14,2};
    std::cout << (c1 == c2) << '\n';  // 0
}
```

Operators

```
#ifndef COMPLEX_T
#define COMPLEX_T

#include <iostream>

struct complex_t
{
    double re;
    double im;

};
bool operator==( complex_t c1, complex_t c2)  // similarly != < etc.
{
    return c1.re == c2.re && c1.im == c2.im;
}

#endif // COMPLEX_T

int main()
{
    complex_t c1{1, 3.14}, c2{3.14,2};
    std::cout << (c1 == c2) << '\n';  // 0
}
```

Operators

```
#ifndef COMPLEX_T
#define COMPLEX_T

#include <iostream>

struct complex_t
{
    double re;
    double im;

};
bool operator==( complex_t c1, complex_t c2)  // similarly != < etc.
{
    return c1.re == c2.re && c1.im == c2.im;
}

#endif // COMPLEX_T

int main()
{
    complex_t c1{1, 3.14}, c2{3.14,2};
    std::cout << std::boolalpha << (c1 == c2) << '\n';  // false
}
```

Constexpr objects in C++11

- Const objects having value known at translation time.
- translation time = compilation time + linking time
- They may have placed to ROM
- Immediately constructed or assigned
- Must contain only literal values, constexpr variables and functions
- The constructor used must be constexpr constructor

Constexpr functions in C++11/14

- Can produce constexpr values when called with compile-time constants.
- Otherwise can run with non-constexpr parameters
- Must not be virtual
- Return type must be literal type
- Parameters must be literal type
- Since C++14 they can be more than a return statement
  - if / else / switch
  - for / ranged-for / while / do-while

Constexpr in C++11

```
enum Flags { good=0, fail=1, bad=2, eof=4 };

constexpr int operator|(Flags f1, Flags f2)
{
  return Flags(int(f1)|int(f2));
}

void f(Flags x)
{
  switch (x)
  {
  case bad:          /* ... */ break;
  case eof:          /* ... */ break;
  case bad|eof:      /* ... */ break;
  default:           /* ... */ break;
  }
}
```

Constexpr in C++11

```
include <cstdio>

size_t constexpr length(const char* str)
{
    return *str ? 1 + length(str + 1) : 0;
}

void f()
{
    printf("%d %d", length("abcd"), length("abcdefgh"));
}
```

- Pattern matching + recursion == Turing complete

Constexpr in C++14

```
#include <iostream>

constexpr int strlen(const char *s)
{
    const char *p = s;
    while ( '\0' != *p )  ++p;
    return p-s;
}
int main()
{
    std::cout << strlen("Hello") << std::endl;
    return 0;
}

constexpr int pow( int base, int exp) noexcept
{
    auto result = 1;
    for (int i = 0; i < exp; ++i) result *= base;
    return result;
}
```

Constexpr in C++20

- Union (also needed for constexpr string)
- Try and catch (throw not allowed, so catch works only runtime)
- dynamic_cast and typeid (since we have virtual functions)
- Constexpr allocation
  - Transient: deallocated before evaluation completes
  - Non-transient: not (yet)
- Virtual calls in constexpr, constexpr virtual funtions/overrides
- Library: constexpr vector and string

Constexpr in C++20

```
// C++20

constexpr auto naiveSum(unsigned int n)
{
    auto p = new int[n];  // transient allocation C++20

    // iota is constexpr in C++20
    std::iota(p, p+n, 1);

    // accumulate is constexpr in C++20
    auto tmp = std::accumulate(p, p+n, 0);

    delete [] p;    // compiler detects delete/delete[] issues
    return tmp;
}
```

Inline functions

```
#ifndef COMPLEX_T
#define COMPLEX_T

#include <iostream>

struct complex_t
{
    double re;
    double im;

};
inline bool operator==( complex_t c1, complex_t c2)  // similarly != < etc.
{
    return c1.re == c2.re && c1.im == c2.im;
}

#endif // COMPLEX_T

int main()
{
    complex_t c1{1, 3.14}, c2{3.14,2};
    std::cout << std::boolalpha << (c1 == c2) << '\n';  // false
}
```

Global declarations in C++

```
#include <iostream>
void f();
            int g0 = 10;
         inline int g1 = 11;
  static inline int g2 = 12;
  extern inline int g3 = 13;

int main()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
  f();
  return 0;
}
```

```
#include <iostream>
                  int g0 = 20;
           inline int g1 = 21;
  static inline int g2 = 22;
  extern inline int g3 = 23;

void f()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
}
```

Global declarations in C++

```
#include <iostream>
void f();
             int g0 = 10;
          inline int g1 = 11;
  static inline int g2 = 12;
  extern inline int g3 = 13;

int main()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
  f();
  return 0;
}
```

/usr/bin/ld: inline2.o:(.data+0x0): multiple definition of `g0'; inline1.o:(.data+0x0): first defined here

```
#include <iostream>
                  int g0 = 20;
          inline int g1 = 21;
  static inline int g2 = 22;
  extern inline int g3 = 23;

void f()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
}
```

Global declarations in C++

```
#include <iostream>
void f();
            int g0 = 10;
        inline int g1 = 11;
  static inline int g2 = 12;
  extern inline int g3 = 13;

int main()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
  f();
  return 0;
}
```

/usr/bin/ld: inline2.o:(.data+0x0): multiple definition of `g0'; inline1.o:(.data+0x0): first defined here

```
#include <iostream>
        extern int g0 = 20;
        inline int g1 = 21;
  static inline int g2 = 22;
  extern inline int g3 = 23;

void f()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
}
```

Global declarations in C++

```
#include <iostream>
void f();
            int g0 = 10;
         inline int g1 = 11;
  static inline int g2 = 12;
  extern inline int g3 = 13;

int main()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
  f();
  return 0;
}
```

```
#include <iostream>
         extern int g0;
         inline int g1 = 21;
  static inline int g2 = 22;
  extern inline int g3 = 23;

void f()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
}
```

Global declarations in C++

```
#include <iostream>
void f();
             int g0 = 10;
          inline int g1 = 11;
  static inline int g2 = 12;
  extern inline int g3 = 13;

int main()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
  f();
  return 0;
}
```

```
#include <iostream>
          extern int g0;
          inline int g1 = 21;
  static inline int g2 = 22;
  extern inline int g3 = 23;

void f()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
}
```

```
&g0 = 10, 0x40406c
&g1 = 21, 0x404060
&g2 = 12, 0x404070
&g3 = 23, 0x404068
&g0 = 10, 0x40406c
&g1 = 21, 0x404060
&g2 = 22, 0x404064
&g3 = 23, 0x404068
```

Global declarations in C++

```
#include <iostream>
void f();
            int g0 = 10;
         inline int g1 = 11;
  static inline int g2 = 12;
  extern inline int g3 = 13;

int main()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
  f();
  return 0;
}
```

```
#include <iostream>
         extern int g0;
         inline int g1 = 21;
  static inline int g2 = 22;
  extern inline int g3 = 23;

void f()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
}
```

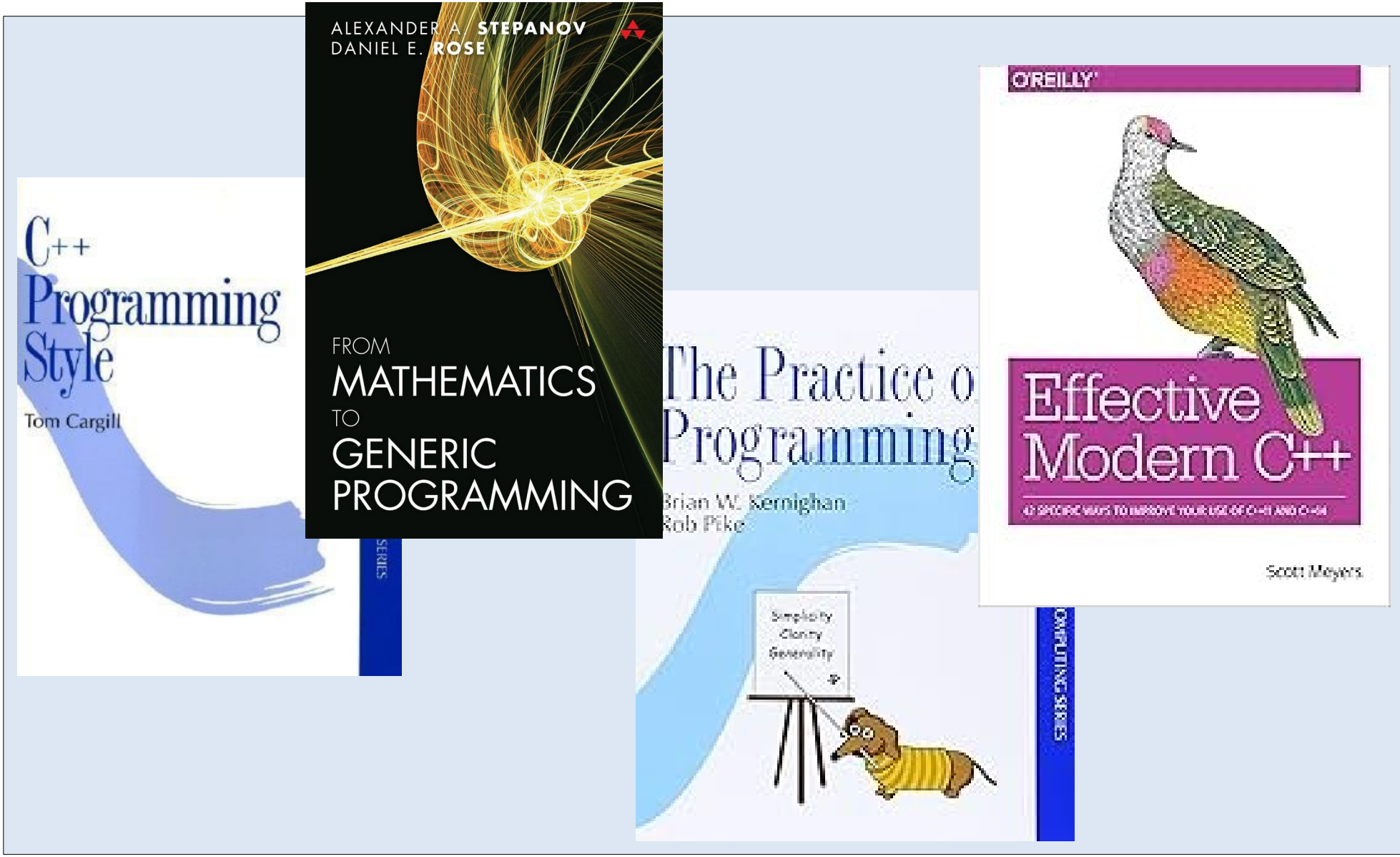
```
&g0 = 10, 0x40406c
&g1 = 11, 0x404060
&g2 = 12, 0x404070
&g3 = 13, 0x404068
&g0 = 10, 0x40406c
&g1 = 11, 0x404060
&g2 = 22, 0x404064
&g3 = 13, 0x404068
```

Linker symbols

```
$ readelf inline1.o

Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS inline1.cpp
     2: 0000000000000040    11 FUNC    LOCAL  DEFAULT    4 _GLOBAL__sub_I_inline1.cp
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    7 _ZL2g2
     4: 0000000000000000     1 OBJECT  LOCAL  DEFAULT    6 _ZStL8__ioinit
     5: 0000000000000000    59 FUNC    LOCAL  DEFAULT    4 __cxx_global_var_init
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    2
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
    10: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
    11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _Z1fv
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZNSolsEPKv
    13: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZNSolsEi
    14: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZNSt8ios_base4InitC1Ev
    15: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZNSt8ios_base4InitD1Ev
    16: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZSt4cout
    17: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZStlsISt11char_traitsIcE
    18: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZStlsISt11char_traitsIcE
    19: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND __cxa_atexit
    20: 0000000000000000     0 NOTYPE  GLOBAL HIDDEN   UND __dso_handle
    21: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    7 g0
    22: 0000000000000000     4 OBJECT  WEAK   DEFAULT   10 g1
    23: 0000000000000000     4 OBJECT  WEAK   DEFAULT   12 g3
    24: 0000000000000000   316 FUNC    GLOBAL DEFAULT    2 main
```

Object oriented Programming



Object oriented Programming

- A class should describe a set of objects
- Use data members for variation in value, virtual functions for variation in behavior
- Independent objects should have independent behavior
- Reduce coupling – minimize interactions between classes
- A constructor should put its object in a well-defined state
- No class is perfect; to narrow a design is better than too broad
- Do not encapsulate essential information – make it available by some means
- Consider default arguments as an alternative to function overloading
- Consider the lifetime (especially for heap objects)
- Use Resource Allocation Is Initialization technique (RAII)

Object oriented Programming

- Define the role of all entity
- Simplicity, clarity
- Write scalable code – understand algorithmic complexity
- Avoid globals
- Hide data
- Know when and how to code for concurrency
- Use constants and constant correctness
- Prefer compile time errors
- Avoid early optimization
- Avoid cyclic dependencies
- Always initialize variables, but should not depend on other compilation units