Gentle introduction to C++

Why use C++?

Máté Ferenc Nagy-Egri
Wigner GPU Lab

Table of Contents

Important disclaimer

- The beginning of this talk will try to clarify:
  - Every language under the aegis of Turing has it’s place
  - Not all languages were created created equal (some are indeed more equal)

Important Disclaimer

- We are evangelists of a given language, although:
  - We are aware it is not the „best" existing one
  - We are keen on educating ourselves in other langs. we see fit
- Although our tone might seem hostile at times:
  - It is ONLY an instrument of conveying emphasis
  - We strive on remaining humble and understanding to the background of newcomers

„Fear leads to anger, anger leads to hate, and hate leads to… suffffferiiiiing.” – Master Yoda

FEAR OF THE UNKNOWN

Hierarchy of disagreement

Explicitly refutes the central point	• Refuting the Central Point
Finds the mistake and explains why it's mistaken using quotes	• Refutation
Contradicts and then backs it up with reasoning and/or supporting evidence	• Counterargument
States the opposite case with little to no reasoning	• Contradiction
Criticizes the tone of the writing without addressing the substance of the argument	• Responding to Tone
Attacks the characteristics or authority of the write without addressing the substance of the argument	• Ad-Hominem
Sounds something like, „you're an idiot"	• Name-calling

The „blub” paradox

A	B	C	D
E	F	G	H
I	J	K	L

The „blub” paradox

- Let’s create a „power spectrum” of all the languages
  - The sorting criteria is the „expressivity” of the language
- For a moment let’s assume such a spectrum exists

The „blub” paradox

- Let’s create a „power spectrum” of all the languages
  - The sorting criteria is the expressivity of the language
- For a moment let’s assume such a spectrum exists
- Choose a programmer adept in an arbitrary (non-extremal) language
  - Let’s call this language „blub”

A B C D F **G** H I J K L

The „blub” paradox

- Our theoretical „blub” programmer can verify the spectrum beneath his language of choice
  - He can tell B is better than A, because of features X & Y which are present in B, but are absent in A

The „blub" paradox

- Our theoretical „blub" programmer can verify the spectrum beneath his language of choice
  - He can tell B is better than A, because of features X & Y which are present in B, but are absent in A
- However, when looking upward in the spectrum, he/she only sees a horde of weird languages
  - Why does feature Z & Q exist, when I can solve all problems without them?

Fear of the unknown

- We can all agree, that writing scientific code in assembler is no longer feasible
- Higher level languages (expressivity) came into existance for a reason
- Programming languages are evolving!
  - Fenomena ranging from „now we can do better” all the way to „seemed like a good idea at the time” all exist

Fear of the unknown

- Cobol, Fortran, C
  - Close to metal languages
  - Hide hardware details
    - uniform address space
    - weak type safety
    - I/O features (operating systems)
  - Imperative in nature (just like punch cards and assembler)
  - Compilers optimize program code to match machine preferences

Fear of the unknown

- C++, D, Go, Rust, C#, Java, etc.
  - „Close to metal" languages
  - Ability to create abstractions
    - Hide implementation details
      - Languages provide varying level of access to low-level features
    - Provide runtime safety
      - Resource management (RAII, garbage collection)
      - Race conditions (parallel programming errors)
      - Turn run-time errors into compile-time errors (!)
    - Expressivity
      - Inheritance and object orientation („is a" versus „has a" relationship)
      - Generic programming (code deduplication versus abstraction)

Fear of the unknown

- OCaml, Haskell, F#, Julia, Idris
  - Academic languages
  - Applied mathematics
    - Expressivity at the forefront
      - Sacrifice runtime performance for a more flexible/expressive type system
      - Cleaner foundations (less industry and more academy)

The „blub" paradox

A	B	C	D	F	G	H	I	J	K	L

The „blub" paradox

asm B C D F G H I J K L

The „blub” paradox

asm → B → C → D → F → G → H → I → J → K → Idris

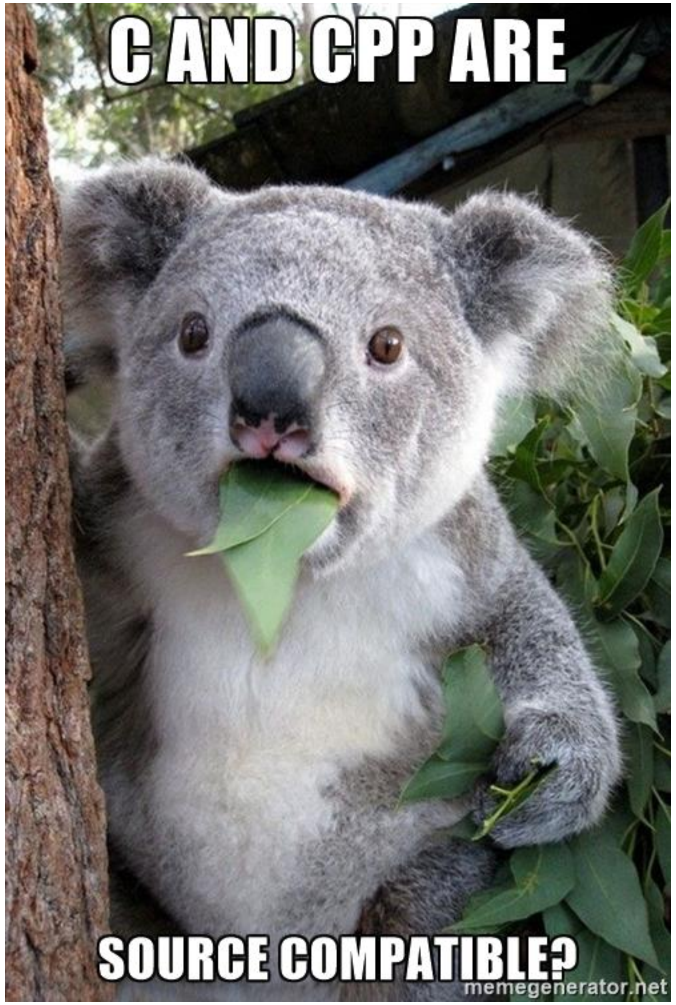
The „blub” paradox

asm → B → C → D → F → **C++** → H → I → J → K → Idris

How to make the transition?

REINTERPRET_CAST<CPP>(C)

And really



- People tend to forget: C++ is source compatible with C
- You don't have to use every fancy feature
  - Tired of string manipulation? Use std::string
  - Want a stateful function? Use a functor
  - Support float and double effortlessly? Sure
- Don't use what you don't want

Source compatibility

```c
// Standard C includes
#include <stdlib.h> // EXIT_SUCCESS
#include <stdio.h> // printf

int main(int argc, const char* argv[], const char* envp[])
{
        int CRT_err = printf("Hello World!");

        if (CRT_err < 0)
                exit(EXIT_FAILURE);

        return EXIT_SUCCESS;
}
```

```
PS C:\Users\Matty\OneDrive\Develop\Wigner\Active\LoMSP> cl.exe /nologo .\src\Minimal.c
Minimal.c
PS C:\Users\Matty\OneDrive\Develop\Wigner\Active\LoMSP> .\Minimal.exe
Hello World!
```

Source compatibility

```
// Standard C includes
#include <stdlib.h> // EXIT_SUCCESS
#include <stdio.h> // printf

int main(int argc, const char* argv[], const char* envp[])
```

```
PS C:\Users\Matty\OneDrive\Develop\Wigner\Active\LoMSP> cl.exe /help
Microsoft (R) C/C++ Optimizing Compiler Version 19.00.24215.1 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

                         C/C++ COMPILER OPTIONS
...
/TP compile all files as .cpp
...
```

```
PS C:\Users\Matty\OneDrive\Develop\Wigner\Active\LoMSP> cl.exe /nologo /TP .\src\Minimal.c
Minimal.c
PS C:\Users\Matty\OneDrive\Develop\Wigner\Active\LoMSP> .\Minimal.exe
Hello World!
```

Source compatibility

```c
// Standard C includes
#include <stdlib.h> // EXIT_SUCCESS
#include <stdio.h> // printf

int main(int argc, const char* argv[], const char* envp[])
{
        int CRT_err = printf("Hello World!");

        if (CRT_err < 0)
                exit(EXIT_FAILURE);

        return EXIT_SUCCESS;
}
```

```
mnagy@MATTY-Z50-75:/mnt/c/Users/Matty/OneDrive/Develop/Wigner/Active/LoMSP$ gcc src/Minimal.c
mnagy@MATTY-Z50-75:/mnt/c/Users/Matty/OneDrive/Develop/Wigner/Active/LoMSP$ ./a.out
Hello World!
```

Source compatibility

```c
// Standard C includes
#include <stdlib.h> // EXIT_SUCCESS
#include <stdio.h> // printf

int main(int argc, const char* argv[], const char* envp[])
{
        int CRT_err = printf("Hello World!");

        if (CRT_err < 0)
                exit(EXIT_FAILURE);

        return EXIT_SUCCESS;
}
```

```
mnagy@MATTY-Z50-75:/mnt/c/Users/Matty/OneDrive/Develop/Wigner/Active/LoMSP$ g++ src/Minimal.c
mnagy@MATTY-Z50-75:/mnt/c/Users/Matty/OneDrive/Develop/Wigner/Active/LoMSP$ ./a.out
Hello World!
```

Source compatibility

- Modulo differences, most notably
  - Variable Length Arrays (on stack) are not supported in C++

```
int a[5]; // OK in C, OK in C++

size_t length = 5;

int* b = (int*)calloc(length, sizeof(int)); // OK in C, OK in C++

int c[length]; // OK in C, ERROR in C++
               //
               // infact, OK in C99, optional in C11
```

- Some other minor differences, but mostly the same

Just how deep the rabbit hole is

And now let's get our hands dirty

Namespaces

- Most C APIs prefix their functions, so that they do not collide with similar functionality of other APIs
  - gsl_integrate_cquad(...)
  - glBegin()
  - clCreateContext(...)
  - etc.
- Function names clearly indicate where features originate from, but is sometimes tedious to write out

Namespaces

```
// Build program file
cl_program build_program_source( cl_context context, const char* source )
{
    ...

    result = clCreateProgramWithSource( context, 1, &source, &length, &CL_err );
    CL_err = clGetContextInfo( context, CL_CONTEXT_NUM_DEVICES, sizeof( cl_uint ), &numDevices, NULL );
    devices = (cl_device_id*)malloc( numDevices * sizeof( cl_device_id ) );
    CL_err = clGetContextInfo( context, CL_CONTEXT_DEVICES, numDevices * sizeof( cl_device_id ), devices,
 NULL );

    // Warnings will be treated like errors, this is useful for debug
    char build_params[] = { "-Werror -cl-std=CL1.0" };
    CL_err = clBuildProgram( result, numDevices, devices, build_params, NULL, NULL );

    ...
}
```

Namespaces

```
// Build program file
cl_program build_program_source( cl_context context, const char* source )
{
    ...

    result = clCreateProgramWithSource( context, 1, &source, &length, &CL_err );
    CL_err = clGetContextInfo( context, CL_CONTEXT_NUM_DEVICES, sizeof( cl_uint ), &numDevices, NULL );
    devices = (cl_device_id*)malloc( numDevices * sizeof( cl_device_id ) );
    CL_err = clGetContextInfo( context, CL_CONTEXT_DEVICES, numDevices * sizeof( cl_device_id ), devices,
 NULL );

    // Warnings will be treated like errors, this is useful for debug
    char build_params[] = { "-Werror -cl-std=CL1.0" };
    CL_err = clBuildProgram( result, numDevices, devices, build_params, NULL, NULL );

    ...
}
```

Namespaces

- C++98 introduces the notion of namespaces, which can be traversed by the operator `::`
- Functions, types and global variables may be placed in namespaces to separate from other APIs
- `using namespace` directives may be used to omit namespace names.
- These directives are scoped
  - In global scope they span the entire translation unit
  - In function/plain scope they are restricted to that scope only

Namespaces

```
int main()
{
    ...

    cl::Buffer buf_x(context, x.begin(), x.end(), true);
    cl::Buffer buf_y(context, x.begin(), x.end(), false);

    cl::CommandQueue queue(context,
                           devices.at(0),
                           cl::QueueProperties::OutOfOrder | cl::QueueProperties::Profiling);

    cl::Event kernel_event;

    kernel_event = vecAdd(cl::EnqueueArgs(queue, cl::NDRange(chainlength)), a, buf_x, buf_y);
    kernel_event.wait();

    ...
}
```

Namespaces

```
int main()
{
    ...

    cl::Buffer buf_x(context, x.begin(), x.end(), true);
    cl::Buffer buf_y(context, x.begin(), x.end(), false);

    cl::CommandQueue queue(context,
                           devices.at(0),
                           cl::QueueProperties::OutOfOrder | cl::QueueProperties::Profiling);

    cl::Event kernel_event;

    kernel_event = vecAdd(cl::EnqueueArgs(queue, cl::NDRange(chainlength)), a, buf_x, buf_y);
    kernel_event.wait();

    ...
}
```

Namespaces

```
int main()
{
    ...
    using namespace cl;
    Buffer buf_x(context, x.begin(), x.end(), true);
    Buffer buf_y(context, x.begin(), x.end(), false);

    CommandQueue queue(context,
                       devices.at(0),
                       QueueProperties::OutOfOrder | QueueProperties::Profiling);

    Event kernel_event;

    kernel_event = vecAdd(EnqueueArgs(queue, NDRange(chainlength)), a, buf_x, buf_y);
    kernel_event.wait();

    ...
}
```

Function overloading

- In C every function has to have a unique name
- Because of this, people end up with functions such as:
  - abs
  - labs
  - llabs
- These all essentially do the same thing, but have a different name
- This is not only tedious, but significantly hinders generic programming
- The standard library in cases provide type-generic versions of such functions #include <tgmath.h> but the macro magic inside, is not for the faint of heart

Function overloading

```c
// Standard C includes
#include <stdlib.h>      // abs, labs
#include <stdio.h>       // printf

unsigned int c_abs(int in) { return (unsigned int)abs(in); }
unsigned long c_labs(long in) { return (unsigned long)labs(in); }
unsigned long long c_llabs(long long in) { return (unsigned long long)llabs(in); }

int main()
{
    long negative = -5;
    unsigned long positive = c_labs(negative);

    printf("Absolute value of %ld is %lu\n", negative, positive);

    return EXIT_SUCCESS;
}
```

Function overloading

```
// Standard C includes
#include <stdlib.h>     // abs, labs
#include <stdio.h>      // printf

unsigned int c_abs(int in) { return (unsigned int)abs(in); }
unsigned long c_labs(long in) { return (unsigned long)labs(in); }
unsigned long long c_llabs(long long in) { return (unsigned long long)llabs(in); }

int main()
{
    long negative = -5;
    unsigned long positive = c_labs(negative);

    printf("Absolute value of %ld is %lu\n", negative, positive);

    return EXIT_SUCCESS;
}
```

Warmbed of copy-paste errors

Function overloading

```c
// Standard C includes
#include <stdlib.h>     // abs, labs
#include <stdio.h>      // printf

typedef long integral; typedef unsigned long natural;

unsigned int c_abs(int in) { return (unsigned int)abs(in); }
unsigned long c_labs(long in) { return (unsigned long)labs(in); }
unsigned long long c_llabs(long long in) { return (unsigned long long)llabs(in); }

int main()
{
    integral negative = -5;
    natural positive = c_labs(negative);

    printf("Absolute value of %ld is %lu\n", negative, positive);

    return EXIT_SUCCESS;
}
```

Try adding some genericity to the code.

Function overloading

```c
// Standard C includes
#include <stdlib.h>      // abs, labs
#include <stdio.h>       // printf

typedef int integral; typedef unsigned int natural;

unsigned int c_abs(int in) { return (unsigned int)abs(in); }
unsigned long c_labs(long in) { return (unsigned long)labs(in); }
unsigned long long c_llabs(long long in) { return (unsigned long long)llabs(in); }

int main()
{
    integral negative = -5;
    natural positive = c_labs(negative);

    printf("Absolute value of %ld is %lu\n", negative, positive);

    return EXIT_SUCCESS;
}
```

What happens if I change the types?

Function overloading

```
// Standard C includes
#include <stdlib.h>      // abs, labs
#include <stdio.h>       // printf

typedef int integral; typedef unsigned int natural;
```

What happens if I change the types?

```
unsigned int c_abs(int in) { return (unsigned int)abs(in); }
unsigned long c_labs(long in) { return (unsigned long)labs(in); }
unsigned long long c_llabs(long long in) { return (unsigned long long)llabs(in); }

int main()
{
    integral negative = -5;
    natural positive = c_labs(negative);

    printf("Absolute value of %ld is %lu\n", negative, positive);

    return EXIT_SUCCESS;
}
```

IT COMPILES!

Function overloading

```
// Standard C includes
#include <stdlib.h>     // abs, labs
#include <stdio.h>      // printf

typedef int integral; typedef unsigned int natural;

unsigned int c_abs(int in) { return (unsigned int)abs(in); }
unsigned long c_labs(long in) { return (unsigned long)labs(in); }
unsigned long long c_llabs(long long in) { return (unsigned long long)llabs(in); }

int main()
{
    integral negative = -5;
    natural positive = c_labs(negative);

    printf("Absolute value of %ld is %lu\n", negative, positive);

    return EXIT_SUCCESS;
}
```

Widening conversion (the better case)

Function overloading

```
// Standard C includes
#include <stdlib.h>      // abs, labs
#include <stdio.h>       // printf

typedef int integral; typedef unsigned int natural;

unsigned int c_abs(int in) { return (unsigned int)abs(in); }
unsigned long c_labs(long in) { return (unsigned long)labs(in); }
unsigned long long c_llabs(long long in) { return (unsigned long long)llabs(in); }

int main()
{
    integral negative = -5;
    natural positive = c_labs(negative);

    printf("Absolute value of %ld is %lu\n", negative, positive);

    return EXIT_SUCCESS;
}
```

We possibly print garbage values to console, %ld might read more than %d

Function overloading

- C++98 allows to „overload" the name of the function to behave differently, depending on the types of its arguments
- Because of this, one can define functions such as:
  - int abs(int)
  - long abs(long)
  - long long abs(long long)
- The internals of these functions might behave differently (and it does) based on input
- The standard library provides this function as [std::abs]()

Function overloading

```
// Standard C/C++ includes
#include <cmath>          // std::abs
#include <stdio.h>        // printf

typedef int integral; typedef unsigned int natural;

unsigned int cpp_abs(int in) { return (unsigned int)std::abs(in); }
unsigned long cpp_abs(long in) { return (unsigned long)std::abs(in); }
unsigned long long cpp_abs(long long in) { return (unsigned long long)std::abs(in); }

int main()
{
    integral negative = -5;
    natural  positive = cpp_abs(negative);

    printf("Absolute value of %ld is %lu\n", negative, positive);

    return EXIT_SUCCESS;
}
```

Function overloading

```
// Standard C/C++ includes
#include <cmath>          // std::abs
#include <stdio.h>        // printf

typedef int integral; typedef unsigned int natural;

unsigned int cpp_abs(int in) { return (unsigned int)std::abs(in); }
unsigned long cpp_abs(long in) { return (unsigned long)std::abs(in); }
unsigned long long cpp_abs(long long in) { return (unsigned long long)std::abs(in); }

int main()
{
    integral negative = -5;
    natural  positive = cpp_abs(negative);

    printf("Absolute value of %ld is %lu\n", negative, positive);

    return EXIT_SUCCESS;
}
```

Do not wiki search std, because here it does not refer to sexually transferable disease, but it denotes the namespace of the C++ STanDard library

Function overloading

```
// Standard C/C++ includes
#include <cmath>          // std::abs
#include <stdio.h>        // printf

typedef int integral; typedef unsigned int natural;

unsigned int cpp_abs(int in) { return (unsigned int)std::abs(in); }
unsigned long cpp_abs(long in) { return (unsigned long)std::abs(in); }
unsigned long long cpp_abs(long long in) { return (unsigned long long)std::abs(in); }

int main()
{
    integral negative = -5;
    natural  positive = cpp_abs(negative);

    printf("Absolute value of %ld is %lu\n", negative, positive);

    return EXIT_SUCCESS;
}
```

Notice how the function names are identical

Therefor the call site need not be changed when the typedefs are changed

Function overloading

```
// Standard C/C++ includes
#include <cmath>          // std::abs
#include <stdio.h>        // printf

typedef int integral; typedef unsigned int natural;

unsigned int cpp_abs(int in) { return (unsigned int)std::abs(in); }
unsigned long cpp_abs(long in) { return (unsigned long)std::abs(in); }
unsigned long long cpp_abs(long long in) { return (unsigned long long)std::abs(in); }

int main()
{
    integral negative = -5;
    natural  positive = cpp_abs(negative);

    printf("Absolute value of %ld is %lu\n", negative, positive);

    return EXIT_SUCCESS;
}
```

However, this part still did not follow the change of typedefs

Operator overloading

- C++98 feature, that allows overloading not just function names, but also the built-in operators to behave differently for both built-in and non-built-in types
- Because writing things like
  - ```
    custom::my_struct a, b;
    custom::my_struct c = a custom::+ b;
    ```
  - would be weird, operators do not reside in namespaces
- Most operators can be overloaded
    ```

Operator overloading

```
// Standard C/C++ includes
#include <cmath>        // std::abs
#include <stdio.h>      // printf

typedef int integral; typedef unsigned int natural;

unsigned int cpp_abs(int in) { return (unsigned int)std::abs(in); }
unsigned long cpp_abs(long in) { return (unsigned long)std::abs(in); }
unsigned long long cpp_abs(long long in) { return (unsigned long long)std::abs(in); }

int main()
{
    integral negative = -5;
    natural  positive = cpp_abs(negative);

    printf("Absolute value of %ld is %lu\n", negative, positive);

    return EXIT_SUCCESS;
}
```

Recall this part was one of our concerns

Operator overloading

```cpp
// Standard C/C++ includes
#include <cmath>          // std::abs
#include <iostream>       // std::cout

typedef int integral; typedef unsigned int natural;

unsigned int cpp_abs(int in) { return (unsigned int)std::abs(in); }
unsigned long cpp_abs(long in) { return (unsigned long)std::abs(in); }
unsigned long long cpp_abs(long long in) { return (unsigned long long)std::abs(in); }

int main()
{
    integral negative = -5;
    natural  positive = cpp_abs(negative);

    std::cout << "Absolute value of " << negative << " is " << positive;

    return EXIT_SUCCESS;
}
```

Operator overloading

```
// Standard C/C++ includes
#include <cmath>         // std::abs
#include <iostream>      // std::cout

typedef int integral; typedef unsigned int natural;

unsigned int cpp_abs(int in) { return (unsigned int)std::abs(in); }
unsigned long cpp_abs(long in) { return (unsigned long)std::abs(in); }
unsigned long long cpp_abs(long long in) { return (unsigned long long)std::abs(in); }

int main()
{
    integral negative = -5;
    natural  positive = cpp_abs(negative);

    std::cout << "Absolute value of " << negative << " is " << positive << std::endl;

    return EXIT_SUCCESS;
}
```

Overloading the binary left-shift operator, we can „push” things into cout, the console out entity. operator<< is by default overloaded for std::ostream (the type of std::cout) and all built-in types.

Function template

- What is a function template?
  - It is NOT a function
  - It is a recipe to create a function
- What is a template function?
  - It is function, that was generated from a template

```
template <typename T> T sq(const T t) { return t*t; };

int main()
{
    int a = sq(5);
}
```

Function template

- What is a function template?
  - It is NOT a function
  - It is a recipe to create a function
- What is a template function?
  - It is function, that was generated from a template

```
template <typename T> T sq(const T t) { return t*t; };

int main()
{
    int a = sq(5);
}
```

This tells us we’re introducing a template

Function template

- What is a function template?
  - It is NOT a function
  - It is a recipe to create a function
- What is a template function?
  - It is function, that was generated from a template

```
template <typename T> T sq(const T t) { return t*t; };

int main()
{
    int a = sq(5);
}
```

Inside the angle brackets, we have a comma separated list of template parameters. In this case, it happens to be the name of a type. We can use this name throughout the function signature and the implementation.

Function template

- What is a function template?
  - It is NOT a function
  - It is a recipe to create a function
- What is a template function?
  - It is function, that was generated from a template

```
template <typename T> T sq(const T t) { return t*t; };

int main()
{
    int a = sq(5);
}
```

When the compiler reaches finishes compiling this part of the code, nothing is done yet. This is not a function, just a recipe to create one.

Function template

- What is a function template?
  - It is NOT a function
  - It is a recipe to create a function
- What is a template function?
  - It is function, that was generated from a template

```cpp
template <typename T> T sq(const T t) { return t*t; };

int main()
{
    int a = sq(5);
}
```

When it reaches the call site, this is when the function is instantiated, meaning that a declaration/definition is generated with the type provided.

Function template

```
// Standard C/C++ includes
#include <cmath>         // std::abs
#include <iostream>      // std::cout

typedef int integral; typedef unsigned int natural;

unsigned int cpp_abs(int in) { return (unsigned int)std::abs(in); }
unsigned long cpp_abs(long in) { return (unsigned long)std::abs(in); }
unsigned long long cpp_abs(long long in) { return (unsigned long long)std::abs(in); }

int main()
{
    integral negative = -5;
    natural  positive = cpp_abs(negative);

    std::cout << "Absolute value of " << negative << " is " << positive << std::endl;

    return EXIT_SUCCESS;
}
```

Function template

```cpp
// Standard C/C++ includes
#include <cmath>          // std::abs
#include <iostream>       // std::cout

typedef int integral; typedef unsigned int natural;

unsigned int cpp_abs(int in) { return (unsigned int)std::abs(in); }
unsigned long cpp_abs(long in) { return (unsigned long)std::abs(in); }
unsigned long long cpp_abs(long long in) { return (unsigned long long)std::abs(in); }

int main()
{
    integral negative = -5;
    natural  positive = cpp_abs(negative);

    std::cout << "Absolute value of " << negative << " is " << positive << std::endl;

    return EXIT_SUCCESS;
}
```

This is not just tedious and repetitive, but also error prone. The names of the functions all match, the implementations due to overloading of the wrapped function also match…

Function template

```
// Standard C/C++ includes
#include <cmath>          // std::abs
#include <iostream>       // std::cout

typedef int integral; typedef unsigned int natural;

template <typename T> T cpp_abs(const T in) { return (T)std::abs(in); };



int main()
{
    integral negative = -5;
    natural  positive = cpp_abs(negative);

    std::cout << "Absolute value of " << negative << " is " << positive << std::endl;

    return EXIT_SUCCESS;
}
```

Variable template

- What is a variable template?
  - It NOT an instance a variable
  - It is a recipe to create a variable
- What is a template variable?
  - It is variable, that was generated from a template
  - Just like other templates, can only be declared at global scope

```
template <typename T> static const T pi = (const T)3.14159265358979323846264338327950;

int main()
{
    float diameter = 2*r*pi<float>;
}
```

Class template

- What is a class template?
  - It is NOT a type
  - It is a recipe to generate a type
- What is a template class?
  - It is a complete type, that was generated from a template

```
template <typename T> struct my_array { T* data; size_t size; }

typedef float real;

int main()
{
    size_t length = 5;
    my_array<real> arr = {(real*)malloc(length * sizeof(real)), length};
}
```

Further on templates

- If you want to know more of templates and how they work, I suggest watching the following tutorial
  - Channel9: Stephan T. Lavavej: Core C++
  - CppCon 2016: Arthur o'Dwyer, Template Normal Programming
  - CppCon 2015: Walter E. Brown, Template Metaprogramming Compendium
- When looking for guides/tutorials on templates, avoid anything with „meta" in it's content
  - Templates != metaprorgramming

References

- Without any decoration, both C and C++ take function parameters „by value”
  - Incurs a copy of the object
  - Modifying the object results in modifying the copy, not the original. This is vexing when
    - One does not wish to pay the extra CPU cycles it takes to make a copy
    - One wishes to write a function that operates on a parameter
    - One wants to emulate multiple return values (C++17 might address this)
- Taking a values pointer in contrast is a solution to all of the above

References

```c
// Standard C includes
#include <stdlib.h>      // malloc, calloc

int main()
{
    size_t length = 8;

    struct arr init = { (double*)malloc(length * sizeof(double)) , length };
    struct arr uninit = { (double*)calloc(length,  sizeof(double)) , length };

    copy(&init, &uninit);

    free(init.data);
    free(uninit.data);
}
```

References

```c
struct arr
{
    double* data;
    size_t size;
};

void copy(const struct arr const* from, struct arr* to)
{
    if (from->size > to->size)
    {
        free(to);
        to->data = (double*)malloc(from->size * sizeof(double));
    }

    for (size_t i = 0; i < from->size; ++i) to->data[i] = from->data[i];
}
```

References

```
struct arr
{
    double* data;
    size_t size;
};

void copy(const struct arr const* from, struct arr* to)
{
    if (from->size > to->size)
    {
        free(to);
        to->data = (double*)malloc(from->size * sizeof(double));
    }

    for (size_t i = 0; i < from->size; ++i) to->data[i] = from->data[i];
}
```

What's wrong with this code?

References

```
struct arr
{
    double* data;
    size_t size;
};

void copy(const struct arr const* from, struct arr* to)
{
    if (from->size > to->size)
    {
        free(to);
        to->data = (double*)malloc(from->size * sizeof(double));
    }

    for (size_t i = 0; i < from->size; ++i) to->data[i] = from->data[i];
}
```

Our function has a precondition we forgot to check: input pointers cannot be null.
We did not check, just blindly dereference and access arr::size member.

References

```
struct arr
{
    double* data;
    size_t size;
};

void copy(const struct arr const* from, struct arr* to)
{
    if (from->size > to->size)
    {
        free(to);
        to->data = (double*)malloc(from->size * sizeof(double));
    }

    for (size_t i = 0; i < from->size; ++i) to->data[i] = from->data[i];
}
```

There is actually a faster implementation with memcpy/memmove

References

```
struct arr
{
    double* data;
    size_t size;
};

void copy(const struct arr const* from, struct arr* to)
{
    if (from->size > to->size)
    {
        free(to);
        to->data = (double*)malloc(from->size * sizeof(double));
    }

    for (size_t i = 0; i < from->size; ++i) to->data[i] = from->data[i];
}
```

All of this must be reimplemented for an array of another type.

References

- A reference is a C++98 entity, that can be interpreted as a restricted pointer
  - Cannot be null (may not remain uninitialized)
  - Cannot be reassigned (once pointing to something, it remains that way)
- Due to restrictions, compilers can emit faster code, and some checks can be safely omitted
- Accessing a varible through a reference incurs no syntactic noise

References

```cpp
// Standard C++ includes
void copy(const std::vector<double>& from, std::vector<double>& to)
{
    if (from.size() > to.size()) to.resize(from.size());

    for (std::size_t i = 0; i < from.size(); ++i) to[i] = from[i];
}

int main()
{
    std::size_t length = 8;

    std::vector<double> init(length, 0);
    std::vector<double> uninit(length);

    copy(init, uninit);
}
```

(Apart from it's name, which was a crime against humanity) Standard vector is the next best thing after ice cream. It is a heap allocated, dynamically sized array that can be resized.

References

```cpp
// Standard C++ includes
void copy(const std::vector<double>& from, std::vector<double>& to)
{
    if (from.size() > to.size()) to.resize(from.size());

    for (std::size_t i = 0; i < from.size(); ++i) to[i] = from[i];
}

int main()
{
    std::size_t length = 8;

    std::vector<double> init(length, 0);
    std::vector<double> uninit(length);

    copy(init, uninit);
}
```

Here, we're invoking two different constructors:

- The first initializing all elements to the second parameter
- The second which default initializes all elements

NOTE: default initialization does not mean zeroing out! It might be zero, might not. (Compilers, build types…)

References

```
// Standard C++ includes
void copy(const std::vector<double>& from, std::vector<double>& to)
{
    if (from.size() > to.size()) to.resize(from.size());

    for (std::size_t i = 0; i < from.size(); ++i) to[i] = from[i];
}

int main()
{
    std::size_t length = 8;

    std::vector<double> init(length, 0);
    std::vector<double> uninit(length);

    copy(init, uninit);
}
```

Not that when we called copy, we did not have to take the address of the variables. Just write their names. Why?

References

```cpp
// Standard C++ includes
void copy(const std::vector<double>& from, std::vector<double>& to)
{
    if (from.size() > to.size()) to.resize(from.size());

    for (std::size_t i = 0; i < from.size(); ++i) to[i] = from[i];
}

int main()
{
    std::size_t length = 8;

    std::vector<double> init(length, 0);
    std::vector<double> uninit(length);

    copy(init, uninit);
}
```

Take a look at the function signature:

- We take our arguments by reference (&)
- First argument is a const&, because we do not modify the source
- Second param is a mutable reference
- Note, that earlier we had const arr const* as the first param. Here we only have one const modifier. Why?
- Because references cannot be reassigned. The reference itself is always const

References

```
// Standard C++ includes
void copy(const std::vector<double>& from, std::vector<double>& to)
{
    if (from.size() > to.size()) to.resize(from.size());

    for (std::size_t i = 0; i < from.size(); ++i) to[i] = from[i];
}

int main()
{
    std::size_t length = 8;

    std::vector<double> init(length, 0);
    std::vector<double> uninit(length);

    copy(init, uninit);
}
```

Take a look at the function signature:

- We take our arguments by reference (&)
- First argument is a const&, because we do not modify the source
- Second param is a mutable reference
- Note, that earlier we had const arr const* as the first param. Here we only have one const modifier. Why?
- Because references cannot be reassigned. The reference itself is always const

References

```
// Standard C++ includes
void copy(const std::vector<double>& from, std::vector<double>& to)
{
    if (from.size() > to.size()) to.resize(from.size());

    for (std::size_t i = 0; i < from.size(); ++i) to[i] = from[i];
}

int main()
{
    std::size_t length = 8;

    std::vector<double> init(length, 0);
    std::vector<double> uninit(length);

    copy(init, uninit);
}
```

Just like before, we resize the array if needed and copy elements one by one.

- Note that we did not have to use operator-> to access member functions of std::vector, just use the references as they were the original variables

References

Pointers are implementation detail

- They are clean cut manifests of memory addresses that the machine thinks in terms of
- Once you start manipulating memory by hand, there's not much the compiler can help you with (safety and performance wise)

References are abstractions

- A reference truly denotes a different name (anchor in code) to the same variable (not region of memory)
- When you talk in terms of references, the compiler knows what you mean and might optimize the reference away completely

Sum up

- So far we've learned how to obtain safety and flexibility through:
  - Function overload to have less function names
  - Operator overload to give prettier interfaces than functions
  - We've created type-generic functions
  - We've created helper variables that may have type according to context
  - New structs that are again type-generic
  - Simplify and optimize our code with references

There are a few things missing to lay the foundations of modern C++

IDIO(MA)TIC C++

Our motto

„Make simple things simpler.”

- Scott Mayers

Catch phrases

- C++ has a few catch phrases that motivates language design
  - Zero overhead abstractions
    - When one abstracts/hides complexity, it should not come at the price of significant performance penalty
  - Don’t pay for what you don’t use
    - If there is a feature of the language or a library you don’t use, it should not incur a performance penalty
  - Novice friendly over expert friendly
    - Facilities for the experts must not make the life of novices harder

Constructors/destructors

- Objects (structs/classes) may need to be initialized before they are ready to be used
  - Initialization usually takes parameters
  - Parameterless initialization is called default construction
    - Unless documented otherwise, default constructed objects are considered to be in an invalid state
- When objects are disposed of (leave scope for eg.) they may need to clean up after themselves
- Copying/moving objects may require special care

```
struct arr
{
    arr() = default;
    arr(const arr& in) : size(in.size)
    {
        memcpy(data, in.data, size * sizeof(double));
    }
    arr(arr&& in)
    {
        std::swap(size, in.size);
        std::swap(data, in.data);
    }
    ~arr() { free(data); }

    double* data;
    size_t size;
};
```

Rule of 3/5/0

- Rule of 3
  - If a class has a copy constructor/assign operator, destructor implemented, it must have all three
- Rule of 5
  - Those above plus move constructor/assign operator
- Rule of 0
  - CTORs/DTORs must only need be implemented if the class itself expresses ownership
  - In all other cases it can safely be defaulted

```cpp
arr& operator=(const arr& in)
{
    size = in.size;
    memcpy(data,
           in.data,
           size * sizeof(double));

    return *this;
}
```

Resource Acquisition Is Initialization

- The winner of the most idiotic programming idiom in the history of computer programming
- Denotes the practice that when objects are
  - constructed, they allocate (take ownership) of all resources they use
  - destructed, they release all resources they use
- C++ makes very strong guarantees upon running CTORs/DTORs, even when process is being terminated, exceptions (see later) occur, etc.

Resource Acquisition Is Initialization

C

```
int main()
{
    size_t length = 8;

    struct arr init = { ... };
    struct arr uninit = { ... };

    copy(&init, &uninit);

    free(init.data);
    free(uninit.data);
}
```

Malloc was explicit

C++

```
int main()
{
    std::size_t length = 8;

    std::vector<double> init(length, 0);
    std::vector<double> uninit(length);

    copy(init, uninit);
}
```

Resource Acquisition Is Initialization

C

```c
int main()
{
    size_t length = 8;

    struct arr init = { ... };
    struct arr uninit = { ... };

    copy(&init, &uninit);

    free(init.data);
    free(uninit.data);
}
```

C++

```cpp
int main()
{
    std::size_t length = 8;

    std::vector<double> init(length, 0);
    std::vector<double> uninit(length);

    copy(init, uninit);
}
```

CTOR mallocs

Resource Acquisition Is Initialization

C

```c
int main()
{
    size_t length = 8;

    struct arr init = { ... };
    struct arr uninit = { ... };

    copy(&init, &uninit);
                              Free was explicit
    free(init.data);
    free(uninit.data);
}
```

C++

```cpp
int main()
{
    std::size_t length = 8;

    std::vector<double> init(length, 0);
    std::vector<double> uninit(length);

    copy(init, uninit);
}
```

Resource Acquisition Is Initialization

C

```
int main()
{
    size_t length = 8;

    struct arr init = { ... };
    struct arr uninit = { ... };

    copy(&init, &uninit);

    free(init.data);
    free(uninit.data);
}
```

C++

```
int main()
{
    std::size_t length = 8;

    std::vector<double> init(length, 0);
    std::vector<double> uninit(length);

    copy(init, uninit);
}
```

DTOR frees

Resource Acquisition Is Initialization

- The term resource is fairly general. It does not only refer to allocated memory. A resource may be:
  - A file handle (open,close)
  - A network socket (listen, close)
  - An API handle (context, event, etc.)
  - An async operation (fork, join)
  - A synchronization primitive (mutex lock, unlock)
- RAII not only provides safety, but greatly simplifies code when dealing with such resources
- The idiom forces the programmer to think about ownership of resources, whose responsibility is the object and who is just an observer?

Typical OpenCL cleanup

C

```c
    // Release OpenCL resources
    for (cl_uint i = 0; i < count; ++i)
        clReleaseDevice(devices[i]);
    clReleaseContext(context);
    clReleaseProgram(program);
    clReleaseKernel(kernel);
    clReleaseMemObject(buf_x);
    clReleaseMemObject(buf_y);
    clReleaseCommandQueue(queue);
    clReleaseEvent(kernel_event);
    // Free host-side memory
    free(x);
    free(y);

    return EXIT_SUCCESS;
}
```

C++

```cpp
    return EXIT_SUCCESS;
}
```

Resource Acquisition Is Initialization

- The STL provides several primitives that encapsulate ownership of heap allocated objects:
  - std::unique_ptr
    - Holds an object with uninque ownership
  - std::shared_ptr
    - Holds an object with shared ownership
  - std::weak_ptr
    - Holds a non-owning, „weak” pointer to an object with shared ownership

Algorithms

```
// Standard C++ includes
void copy(const std::vector<double>& from, std::vector<double>& to)
{
    if (from.size() > to.size()) to.resize(from.size());

    for (std::size_t i = 0; i < from.size(); ++i) to[i] = from[i];
}

int main()
{
    std::size_t length = 8;

    std::vector<double> init(length, 0);
    std::vector<double> uninit(length);

    copy(init, uninit);
}
```

What’s wrong with this code?

Algorithms

```
// Standard C++ includes
void copy(const std::vector<double>& from, std::vector<double>& to)
{
    if (from.size() > to.size()) to.resize(from.size());

    for (std::size_t i = 0; i < from.size(); ++i) to[i] = from[i];
}

int main()
{
    std::size_t length = 8;

    std::vector<double> init(length, 0);
    std::vector<double> uninit(length);

    copy(init, uninit);
}
```

What's wrong with this code?

This is code we never should've written in the first place.

Algorithms

```
// Standard C++ includes







int main()
{
    std::size_t length = 8;

    std::vector<double> init(length, 0);
    std::vector<double> uninit(length);

    std::copy(init.data(), init.data() + init.size(), uninit.data());
}
```

This is the kind of code we should've written in the first place*. Surely copying an array into another array is common enough for the STL to provide a facility.

*: Wait for it…

Algorithms

- Algorithms are common building blocks of software
- They are programming primitives that operate on a range of elements
  - What is a range? Zero to some non-infinite number of elements. (hand waving definition)
- Some algorithms modify the range(s) they operate on, others just observe it/them
- For a comprehensive list of common operations you do not have to write by hand, see here.

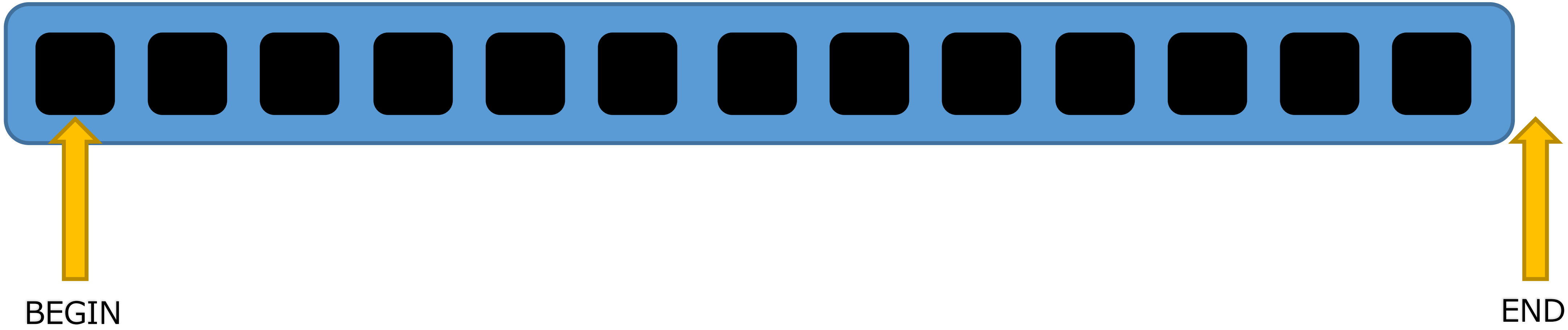
No joking!

SEE HERE!

Iterators

- Throughout the sample codes of algorithms, one will find weird .begin() and .end() pairs on STL containers
- Those return iterator objects
  - Iterators traverse containers
  - They act like pointers, but are objects
  - They provide safety for traversing containers
  - Through operator overloading they truly behave like pointers
- Iterators link algorithms to containers in an $\mathcal{O}(N)$ fashion as opposed to an $\mathcal{O}(N^2)$ fashion

Iterators



Iterators

Sequence
- std::array
- std::vector
- std::deque
- std::list

Associative
- std::set
- std::map
- std::multiset
- std::multimap

Adapted
- std::stack
- std::queue
- std::priority_queue

Algorithms

```
// Standard C++ includes







int main()
{
    std::size_t length = 8;

    std::vector<double> init(length, 0);
    std::vector<double> uninit(length);

    std::copy(init.data(), init.data() + init.size(), uninit.data());
}
```

This code should instead look like…

Algorithms

```cpp
// Standard C++ includes







int main()
{
    std::size_t length = 8;

    std::vector<double> init(length, 0);
    std::vector<double> uninit(length);

    std::copy(init.cbegin(), init.cend(), uninit.begin());
}
```

This code should instead look like…

Algorithms

- And why exactly is this algo mumbo jumbo good for me?
  - It makes code more readable
  - It makes code more maintainable
  - It makes code more portable
  - It allows faster coding (no reinventing the wheel)
  - It allows for faster code (optimizations you didn’t think of)

Algorithms

- Faster, bahh... nothing is faster than optimized C
  - Yes, optimized assembler
  - But didn't we argue earlier that writing optimized assembler is no longer feasible?
  - Isn't it possible, that writing optimized C isn't either?
  - Didn't C++ state it is capable of „Zero Cost Abstractions"?

Algorithms

```
struct arr
{
    double* data;
    size_t size;
};

void copy(const struct arr const* from, struct arr* to)
{
    if (from->size > to->size)
    {
        free(to);
        to->data = (double*)malloc(from->size * sizeof(double));
    }

    for (size_t i = 0; i < from->size; ++i) to->data[i] = from->data[i];
}
```

There is actually a faster implementation with memcpy/memmove
Did I actually stop and think about it?

Algorithms

```
struct arr
{
    double* data;
    size_t size;
};

void copy(const struct arr const* from, struct arr* to)
{
    if (from->size > to->size)
    {
        free(to);
        to->data = (double*)malloc(from->size * sizeof(double));
    }

    for (size_t i = 0; i < from->size; ++i) to->data[i] = from->data[i];
}
```

All of this must be reimplemented for an array of another type.

What if that type happens to be a RAII type?

Algorithms

```
struct arr
{
    double* data;
    size_t size;
};

void copy(const struct arr const* from, struct arr* to)
{
    if (from->size > to->size)
    {
        free(to);
        to->data = (double*)malloc(from->size * sizeof(double));
    }

    for (size_t i = 0; i < from->size; ++i) to->data[i] = from->data[i];
}
```

All of this must be reimplemented for an array of another type.

What if that type happens to be a RAII type?

wasted

Algorithms

```
struct side_effect
{
    side_effect() : _a(0) { ++_a; }
    side_effect(const side_effect& rhs) : _a(rhs._a) { ++_a; }
    side_effect(side_effect&& rhs) : _a(rhs._a) { ++_a; }
    ~side_effect() = default;

    side_effect& operator=(const side_effect& rhs) { ++(_a = rhs._a); return *this; }
    side_effect& operator=(side_effect&& rhs) { ++(_a = std::move(rhs._a)); return *this; }

    int _a;
};
```

This class does nothing special, other than not being [TriviallyCopyable]

Algorithms

```cpp
int main()
{
    std::vector<int> a(10, 1), b(10);
    std::vector<side_effect> c(10), d(10);

    std::copy(a.cbegin(), a.cend(), b.begin());

    std::copy(c.cbegin(), c.cend(), d.begin());

    return 0;
}
```

What are my expectations of the implementation of std::copy?

- I would expect the first to invoke memmove, because I know that is what I would write in plain C because that compiles to special assembler instructions that copy data
- I would expect the second to compile to a plain for loop, because memmove will result in erronous behavior

Algorithms

```
int main()
{
    std::vector<int> a(10, 1), b(10);
    std::vec
    
    std::cop
    
    std::cop
    
    return 0
}
```

What are my expectations of the implementation of std::copy?

- I would expect the first to invoke memmove, because I know

e that compiles to

ta

lain for loop,

behavior

```
template<class _InIt,
	class _OutIt> inline
	_OutIt _Copy_memmove(_InIt _First, _InIt _Last,
		_OutIt _Dest)
	{	// implement copy-like function as memmove
	const char * const _First_ch = reinterpret_cast<const char *>(_First);
	const char * const _Last_ch = reinterpret_cast<const char *>(_Last);
	char * const _Dest_ch = reinterpret_cast<char *>(_Dest);
	const size_t _Count = _Last_ch - _First_ch;
	_CSTD memmove(_Dest_ch, _First_ch, _Count);
	return (reinterpret_cast<_OutIt>(_Dest_ch + _Count));
	}
```

Algorithms

```
int main()
{
    std::vector<int> a(10, 1), b(10);
    std::vec

    std::cop

    std::cop

    return 0
}
```

What are my expectations of the implementation of std::copy?

- I would expect the first to invoke memmove, because I know
e that compiles to
ta
plain for loop,
behavior

```
template<class _InIt,
    class _OutIt> inline
    _OutIt _Copy_memmove(_InIt _First, _InIt _Last,
        _OutIt _Dest)
    {    // implement copy-like function as memmove
    const char * const _First_ch = reinterpret_cast<const char *>(_First);
    const char * const _Last_ch = reinterpret_cast<const char *>(_Last);
    char * const _Dest_ch = reinterpret_cast<char *>(_Dest);
    const size_t _Count = _Last_ch - _First_ch;
    _CSTD memmove(_Dest_ch, _First_ch, _Count);
    return (reinterpret_cast<_OutIt>(_Dest_ch + _Count));
    }
```

Algorithms

```cpp
int main()
{
    std::vector<int> a(10, 1), b(10);
    std::vector<side_effect> c(10), d(10);

    std::copy(a.cbegin(), a.cend(), b.begin());

    std::copy(c.cbegin(), c.cend(), d.begin());

    return 0;
}
```

What are my expectations of the implementation of std::copy?

- I would expect the first to invoke memmove, because I know that is what I would write in plain C because that compiles to special assembler instructions that copy data
- I would expect the second to compile to a plain for loop, because memmove will result in erronous behavior

Algorithms

```
int main()
{
    std::vector<int> a(10, 1), b(10);
    std::vector<side_effect> c(10), d(10);

    std::cop

    std::cop

    return 0
}
```

```
template<class _InIt,
    class _OutIt> inline
    _OutIt _Copy_unchecked1(_InIt _First, _InIt _Last,
        _OutIt _Dest, _General_ptr_iterator_tag)
    {    // copy [_First, _Last) to [_Dest, ...), arbitrary iterators
    for (; _First != _Last; ++_Dest, (void)++_First)
        *_Dest = *_First;
    return (_Dest);
    }
```

What are my expectations of the implementation of std::copy?

- I would expect the first to invoke memmove, because I know that is what I would write in plain C because that compiles to
  ta
  lain for loop,
  ehavior

Sum up

- So far we've learned how to obtain safety and flexibility through:
  - Function overload to have less function names
  - Operator overload to give prettier interfaces than functions
  - We've created type-generic functions
  - We've created helper variables that may have type according to context
  - New structs that are again type-generic
  - Simplify and optimize our code with references

Sum up

- And also:
  - C++ extended C in ways which allow the programmer to
    - stop thinking less about implementation details and more about the actual problem he/she wants solved
    - abstract various aspects of coding (resource management, reoccuring control flow patterns) into reusable primitives that adapt their behavior according to calling context
  - and this is just the tip of the iceberg
    - I have consciously omitted the more advanced stuff, such as template meta-programming, which drives most of the STL and other high-quality libraries

Questions?

THANK YOU FOR YOUR ATTENTION