

# C++ for C Programmers

CS 211

Winter 2020

# Road map

- C versus C++
- Headers
- I/O
- Pass-by-reference
- Dynamic memory and vectors

# C versus C++

# The Good

<b>C</b>	<b>C++</b>
you must call free yourself	helpfully frees data when owning objects go out of scope
need a different name for every function	can reuse function names for different types
operators like + and == work only on built-in types	you can make operators work on user-defined types

# The Bad

<b>C</b>	<b>C++</b>
you know when things are freed	things get freed when you don't expect it
you know what function you're calling	you have to figure out the types to figure out which overload is selected
you know that + means "plus"	you know that + takes two arguments

# Headers

# The standard C headers are renamed

C-style:

```
#include <ctype.h>  
#include <math.h>  
#include <stdio.h>  
#include <string.h>
```

## The standard C headers are renamed

C-style:

```
#include <ctype.h>  
#include <math.h>  
#include <stdio.h>  
#include <string.h>
```

C++-style:

```
#include <cctype>  
#include <cmath>  
#include <cstdio>  
#include <cstring>
```



## The standard C headers are renamed

C-style:

```
#include <ctype.h>  
#include <math.h>  
#include <stdio.h>  
#include <string.h>
```

C++-style:

```
#include <cctype>  
#include <cmath>  
#include <cstdio>  
#include <cstring>
```

Real C++:

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

I/O

## I/O got easier and safer

```
#include <iostream>
```

```
int main()
```

```
{  
    std::cout << "Enter two numbers:\n";  
  
    double x, y;  
    std::cin >> x >> y;  
    if (!std::cin) {  
        std::cerr << "Could not read numbers!\n";  
        return 1;  
    }  
  
    std::cout << x << " * " << y  
              << " == " << x * y << "\n";  
}
```

# Pass-by-reference

## C is completely pass-by-value

```
void f(int x, int* p) { ... }
```

In C, every variable names its own object:

- x stands for 4 bytes, not overlapping with any other variable's object
- p stands for 8 bytes, not overlapping with any other variable's object

C *simulates* pass-by-reference by letting you pass pointers, but you are still passing a value (a pointer value)

## C++ has pass-by-reference as well

```
void f(int x, int* p, int& r) { ... }
```

- x and p are as in C
- r refers to some other, existing `int` object
- r is borrowed and cannot be `nullptr`

Use r like an ordinary `int`—no need to dereference

## C++ reference example: increment

```
#include <cassert>
```

```
void inc_p(int* p)  
{  
    *p += 1;  
}
```

```
void c_style(void)  
{  
    int x = 0;  
    inc_p(&x);  
    assert( x == 1 );  
}
```

## C++ reference example: increment

```
#include <cassert>
```

```
void inc_p(int* p)
{
    *p += 1;
}
```

```
void c_style(void)
{
    int x = 0;
    inc_p(&x);
    assert( x == 1 );
}
```

```
#include <check.h>
```

```
void inc_r(int& r)
{
    r += 1;
}
```

```
TEST_CASE("C++-style")
{
    int x{0};
    inc_r(x);
    CHECK( x == 1 );
}
```



## C++ reference example: swap

```
void swap_p(int* p, int* q) { ... }
```

```
void swap_r(int& r, int& s)
{
    int temp = r;
    r = s;
    s = temp;
}
```

## C++ reference example: swap

```
void swap_p(int* p, int* q) { ... }
```

```
void swap_r(int& r, int& s)
{
    int temp = r;
    r = s;
    s = temp;
}
```

```
TEST_CASE("C++-style_swap")
{
    int x = 3, y = 4;
    swap_r(x, y);
    CHECK( x == 4 ); CHECK( y == 3 );
}
```

## C++ reference example: swap

```
void swap_p(int* p, int* q) { ... }
```

```
void swap_r(int& r, int& s)
{
    int temp = r;
    r = s;
    s = temp;
}
```

```
TEST_CASE("C++-style_swap")
{
    int x = 3, y = 4;
    swap_r(x, y);
    CHECK( x == 4 ); CHECK( y == 3 );
}
```

(swap\_r is std::swap<int>.)

## C++ references *desugar* to pointers

- Replace every declaration  $T\& x$  with  $T* xp$ .

## C++ references *desugar* to pointers

- Replace every declaration `T& x` with `T* xp`.
- Replace every initialization `T& x = e;` with `T* xp = &e;`.

## C++ references *desugar* to pointers

- Replace every declaration `T& x` with `T* xp`.
- Replace every initialization `T& x = e;` with `T* xp = &e;`.
- Replace every use of `x` with `*xp`.

## C++ references *desugar* to pointers

- Replace every declaration `T& x` with `T* xp`.
- Replace every initialization `T& x = e;` with `T* xp = &e;`.
- Replace every use of `x` with `*xp`.

```
void swap(int& r, int& s)
{
    int temp = r;
    r = s;
    s = temp;
}
```

## C++ references *desugar* to pointers

- Replace every declaration `T& x` with `T* xp`.
- Replace every initialization `T& x = e;` with `T* xp = &e;`.
- Replace every use of `x` with `*xp`.

```
void swap(int& r, int& s)
```

```
{  
    int temp = r;  
    r = s;  
    s = temp;
```

```
}
```

*// becomes*

```
void swap(int* rp, int* sp)
```

```
{  
    int temp = *rp;  
    *rp = *sp;  
    *sp = temp;
```

```
}
```



## C++ references *desugar* to pointers

- Replace every declaration `T& x` with `T* xp`.
- Replace every initialization `T& x = e;` with `T* xp = &e;`.
- Replace every use of `x` with `*xp`.

```
void swap(int& r, int& s)           swap(x, y);
{
    int temp = r;
    r = s;
    s = temp;
}
```

*// becomes*

```
void swap(int* rp, int* sp)       swap(&x, &y);
{
    int temp = *rp;
    *rp = *sp;
    *sp = temp;
}
```

## Example: alternative swap definition

Does this work?

```
void alt_swap(int& r, int& s)
{
    int& temp = r;
    r = s;
    s = temp;
}
```

## Example: alternative swap definition

Does this work?

```
void alt_swap(int& r, int& s)
{
    int& temp = r;
    r = s;
    s = temp;
}
```

*// becomes*

```
void alt_swap(int* rp, int* sp)
{
    int* tempp = &*rp;
    *rp = *sp;
    *sp = *tempp;
}
```

# Dynamic memory and vectors

## Dynamic memory allocation

The old C way:

```
int* p = malloc(n * sizeof(int));  
if (!p) ...  
for (size_t i = 0; i < n; ++i) p[i] = 0;  
  
...  
  
free(p);
```

## Dynamic memory allocation

The old C way:

```
int* p = malloc(n * sizeof(int));  
if (!p) ...  
for (size_t i = 0; i < n; ++i) p[i] = 0;  
  
...  
  
free(p);
```

Manually in C++:

```
int* p = new int[n];  
  
...  
  
delete [] p;
```

## Dynamic memory allocation

The old C way:

```
int* p = malloc(n * sizeof(int));  
if (!p) ...  
for (size_t i = 0; i < n; ++i) p[i] = 0;  
  
...  
  
free(p);
```

Manually in C++:

```
int* p = new int[n];  
  
...  
  
delete [] p;
```

Automatically in C++:

```
std::vector<int> v(n, 0);  
  
...
```

## Using `std::vector` (1/3)

```
#include <check.h>
```

```
#include <vector>
```

```
TEST_CASE("vector_creation_and_access")
```

```
{
```

```
    std::vector<int> v1{ 2, 4, 6, 8 };
```

```
    std::vector<double> v2(10, 3.5);
```

```
    CHECK( v1.size() == 4 );
```

```
    CHECK( v2.size() == 10 );
```

```
    CHECK( v1[1] == 4 );
```

```
    CHECK( v2[1] == 3.5 );
```

```
    v1[1] = 15;
```

```
    CHECK( v1[1] == 15 );
```

```
}
```



## Using `std::vector` (2/3)

```
using VI = std::vector<int>;
```

```
TEST_CASE("growing_and_shrinking")
```

```
{  
    VI v;  
  
    CHECK( v == VI{} );  
    v.push_back(2);  
    CHECK( v == VI{2} );  
    v.push_back(5);  
    v.push_back(9);  
    CHECK( v == VI{2, 5, 9} );  
  
    v.pop_back();  
    CHECK( v == VI{2, 5} );  
}
```

## Using `std::vector` (3/3)

```
#include <stdexcept>
```

```
TEST_CASE("bounds_checking_(or_not)")
```

```
{
```

```
    std::vector<int> v{2, 3, 4};
```

```
        CHECK(v.at(2) == 4);
```

```
        v.at(2) = 8;
```

```
        CHECK(v.at(2) == 8);
```

```
        CHECK_THROWS_AS(v.at(3), std::out_of_range);
```

```
        v[10] = 12;           // UB!
```

```
        CHECK( v[10] == 12 ); // also UB!
```

```
}
```

## std::vector is passed by value...

```
void inc_vec(std::vector<int> v)
{
    for (size_t i = 0; i < v.size(); ++i)
        ++v[i];
}
```

```
TEST_CASE("vector_passed_by_value")
{
    VI v{2, 3, 4};
    inc_vec(v);
    CHECK( v == VI{3, 4, 5} ); // FAILS!
}
```

## ...unless passed by reference

```
void inc_vec(std::vector<int>& v)
{
    for (size_t i = 0; i < v.size(); ++i)
        ++v[i];
}
```

```
TEST_CASE("vector_passed_by_reference")
{
    VI v{2, 3, 4};
    inc_vec(v);
    CHECK( v == VI{3, 4, 5} ); // SUCCEEDS!
}
```

## Easier (and more generic) iteration

```
double sum_vec(std::vector<double> const& v)
{
    double result = 0;
    for (double d : v) result += d;
    return result;
}
```

## Easier (and more generic) iteration

```
double sum_vec(std::vector<double> const& v)
{
    double result = 0;
    for (double d : v) result += d;
    return result;
}
```

```
void dec_vec(std::vector<int> &v)
{
    for (int z : v) --z;
}
```

## Easier (and more generic) iteration

```
double sum_vec(std::vector<double> const& v)
{
    double result = 0;
    for (double d : v) result += d;
    return result;
}
```

```
void dec_vec(std::vector<int> &v)
{
    for (int& z : v) --z;
}
```

## More `std::vector<T>` operations

- `bool empty() const;`
- `T& front();`
- `T& back();`
- `T const& front() const;`
- `T const& back() const;`
- `void clear();`
- `void resize(size_t count, T const& fill);`
- `void resize(size_t count);`

See API reference for more:

<https://en.cppreference.com/w/cpp/container/vector>



– Next: Access Control —