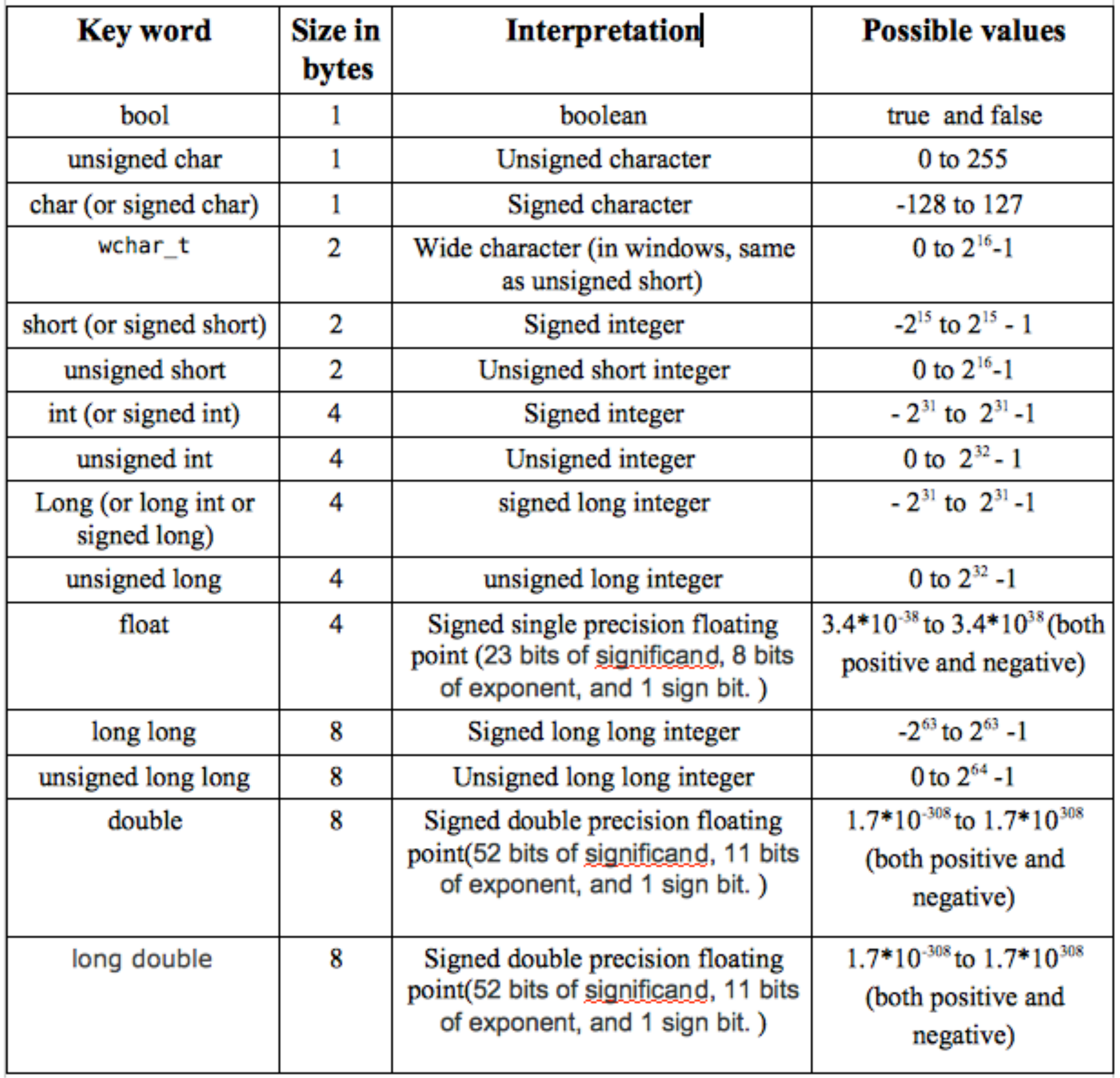
Data Types, Functions, Variable Scope, Namespaces, Pointers and References, Constants

Data Types



Key word	Size in bytes	Interpretation	Possible values
bool	1	boolean	true and false
unsigned char	1	Unsigned character	0 to 255
char (or signed char)	1	Signed character	-128 to 127
wchar_t	2	Wide character (in windows, same as unsigned short)	0 to 2^{16}-1
short (or signed short)	2	Signed integer	-2^{15} to 2^{15} - 1
unsigned short	2	Unsigned short integer	0 to 2^{16}-1
int (or signed int)	4	Signed integer	- 2^{31} to 2^{31} -1
unsigned int	4	Unsigned integer	0 to 2^{32} - 1
Long (or long int or signed long)	4	signed long integer	- 2^{31} to 2^{31} -1
unsigned long	4	unsigned long integer	0 to 2^{32} -1
float	4	Signed single precision floating point (23 bits of significand, 8 bits of exponent, and 1 sign bit.)	3.4*10^{-38} to 3.4*10^{38} (both positive and negative)
long long	8	Signed long long integer	-2^{63} to 2^{63} -1
unsigned long long	8	Unsigned long long integer	0 to 2^{64} -1
double	8	Signed double precision floating point(52 bits of significand, 11 bits of exponent, and 1 sign bit.)	1.7*10^{-308} to 1.7*10^{308} (both positive and negative)
long double	8	Signed double precision floating point(52 bits of significand, 11 bits of exponent, and 1 sign bit.)	1.7*10^{-308} to 1.7*10^{308} (both positive and negative)

- Size is architecture dependent
- This table is for a typical 32-bit architecture
- `int` usually has size of “one word” on a given architecture, so `int` and `long int` may differ in size
- For the 4 integer types:

  `size(short) <= size(int) <= size(long) <= size(long long)`
- Further:

  `size(char) <= size(short)`

Data Types

Based on `examples/02/SizeOfTypes.cpp`

```
#include <iostream>
using namespace std;

int main() {
  char        aChar = 'c'; // char
  bool        aBool = true; // boolean
  short       aShort = 33; // short
  long        aLong  = 123421; // long
  int         anInt = 27; // integer
  float       aFloat = 1.043; // single precision
  double      aDbl = 1.243e-234; // double precision
  long double aLD = 0.432e245; // double precision

  cout << "char aChar = " << aChar << "\tsizeof(" << "char" << "): " << sizeof(aChar) << endl;
  cout << "bool aBool = " << aBool << "\tsizeof(" << "bool" << "): " << sizeof(aBool) << endl;
  cout << "short aShort = " << aShort << "\tsizeof(" << "short" << "): " << sizeof(aShort) << endl;
  cout << "long aLong = " << aLong << "\tsizeof(" << "long" << "): " << sizeof(aLong) << endl;
  cout << "int aInt = " << anInt << "\tsizeof(" << "int" << "): " << sizeof(anInt) << endl;
  cout << "float aFloat = " << aFloat << "\tsizeof(" << "float" << "): " << sizeof(aFloat) << endl;
  cout << "double aDbl = " << aDbl << "\tsizeof(" << "double" << "): " << sizeof(aDbl) << endl;
  cout << "long double aLD = " << aLD << "\tsizeof(" << "long double" << "): " << sizeof(aLD) << endl;


  return 0;
}
```

```
$ g++ -o SizeOfTypes  SizeOfTypes.cpp
$ ./SizeOfTypes
char aChar = c                 sizeof(char): 1
bool aBool = 1                 sizeof(bool): 1
short aShort = 33              sizeof(short): 2
long aLong = 123421            sizeof(long): 8
int aInt = 27                  sizeof(int): 4
float aFloat = 1.043           sizeof(float): 4
double aDbl = 1.243e-234       sizeof(double): 8
long double aLD = 4.32e+244    sizeof(long double): 8
```

Data Types

Based on `examples/02/SizeOfTypes.cpp`

```cpp
#include <iostream>
using namespace std;

int main() {
  char        aChar = 'c'; // char
  bool        aBool = true; // boolean
  short       aShort = 33; // short
  long        aLong  = 123421; // long
  int         anInt = 27; // integer
  float       aFloat = 1.043; // single precision
  double      aDbl = 1.243e-234; // double precision
  long double aLD = 0.432e245; // double precision

  cout << "char aChar = " << aChar << "\tsizeof(" << "char" << "): " << sizeof(aChar) << endl;
  cout << "bool aBool = " << aBool << "\tsizeof(" << "bool" << "): " << sizeof(aBool) << endl;
  cout << "short aShort = " << aShort << "\tsizeof(" << "short" << "): " << sizeof(aShort) << endl;
  cout << "long aLong = " << aLong << "\tsizeof(" << "long" << "): " << sizeof(aLong) << endl;
  cout << "int aInt = " << anInt << "\tsizeof(" << "int" << "): " << sizeof(anInt) << endl;
  cout << "float aFloat = " << aFloat << "\tsizeof(" << "float" << "): " << sizeof(aFloat) << endl;
  cout << "double aDbl = " << aDbl << "\tsizeof(" << "double" << "): " << sizeof(aDbl) << endl;
  cout << "long double aLD = " << aLD << "\tsizeof(" << "long double" << "): " << sizeof(aLD) << endl;


  return 0;
}
```

```
$ g++ -o SizeOfTypes  SizeOfTypes.cpp
$ ./SizeOfTypes
char aChar = c                  sizeof(char): 1
bool aBool = 1                  sizeof(bool): 1
short aShort = 33               sizeof(short): 2
long aLong = 123421             sizeof(long): 8
int aInt = 27                   sizeof(int): 4
float aFloat = 1.043            sizeof(float): 4
double aDbl = 1.243e-234        sizeof(double): 8
long double aLD = 4.32e+244     sizeof(long double): 8
```

- Later, we will see how to introduce new data types (with classes)

Functions

Based on `examples/02/SimpleFuncs1.cpp`

A function is a set of operations to be executed

- Typically takes some input
- Usually returns a value: if it does not, it is `void`
- Must be declared before being invoked (what happens if you swap the order of `times_pi` and `main`?)

```
$ g++ -o SimpleFuncs1  SimpleFuncs1.cpp
$ ./SimpleFuncs1
What number do you want to multiply by pi?
2
Your number times pi is: 6.28
```

```
#include <iostream>

// Function with a single input
double times_pi(double a) {
    return 3.14*a;
}

// Input-free void function
void question() {
    std::cout << "What number do you want to multiply by pi? " << std::endl;
}

int main() {

    double a;

    question();
    std::cin >> a;

    std::cout << "Your number times pi is: " << times_pi(a) << std::endl;

    return 0;
}
```

Functions

Based on `examples/02/SimpleFuncs2.cpp`

```
#include <iostream>

// Function with a single input
double times_pi(double a) {
    return 3.14*a;
}

// Declare to compiler question() is a void function
extern void question();

int main() {

    double a;

    question();
    std::cin >> a;

    std::cout << "Your number times pi is: " << times_pi(a) << std::endl;

    return 0;
}

// Now implement/define question()
void question() {
    std::cout << "What number do you want to multiply by pi? " << std::endl;
}
```

- Functions can be defined elsewhere however

- `extern` tells the compiler that a void function called `question` exists somewhere. It is not the compilers job to know where it exists, it just needs to know the type and name so it knows how to use it. The linker will resolve all of the references of `question` to the one definition that it finds in one of the compiled source files.

Compile and run this code: does it behave as `SimpleFuncs1`?

Scope of Variables

Based on `examples/02/SimpleScope.cpp`

```cpp
#include <iostream>
using namespace std;

int main() {

    double x = 1.2;

    cout << "in main before scope, x: " << x << endl;

    for(int i=0; i<3; ++i) { // just a local scope
        cout << "--> i: " << i << endl;

        x++;
        cout << "in local scope before int, x: " << x << endl;

        int x = 4;
        cout << "in local scope after int, x: " << x << endl;
    }

    //This will not work: it is outside the scope of i
    // cout << i << endl;
    cout << "in main after local scope, x: " << x << endl;

    return 0;
}
```

The scope of a name is the block of program where the name is valid and can be used

- A block is delimited by `{}`
- It can be the body of a function or a simple scope defined by the user with `{}`

Scope of Variables

Based on **`examples/02/SimpleScope.cpp`**

```
#include <iostream>
using namespace std;

int main() {

    double x = 1.2;

    cout << "in main before scope, x: " << x << endl;

    for(int i=0; i<3; ++i) { // just a local scope
        cout << "--> i: " << i << endl;

        x++;
        cout << "in local scope before int, x: " << x << endl;

        int x = 4;
        cout << "in local scope after int, x: " << x << endl;
    }

    //This will not work: it is outside the scope of i
    // cout << i << endl;
    cout << "in main after local scope, x: " << x << endl;

    return 0;
}
```

The scope of a name is the block of program where the name is valid and can be used

- A block is delimited by `{}`
- It can be the body of a function or a simple scope defined by the user with `{}`

➡ Changed value of x from main scope

➡ Defined new variable x in this scope

➡ Back to the main scope

Scope of Variables

Based on `examples/02/SimpleScope.cpp`

```cpp
#include <iostream>
using namespace std;

int main() {

    double x = 1.2;

    cout << "in main before scope, x: " << x << endl;

    for(int i=0; i<3; ++i) { // just a local scope
        cout << "--> i: " << i << endl;

        x++;
        cout << "in local scope before int, x: " << x << endl;

        int x = 4;
        cout << "in local scope after int, x: " << x << endl;
    }

    //This will not work: it is outside the scope of i
    // cout << i << endl;
    cout << "in main after local scope, x: " << x << endl;

    return 0;
}
```

The scope of a name is the block of program where the name is valid and can be used

- A block is delimited by `{}`
- It can be the body of a function or a simple scope defined by the user with `{}`

➡ Changed value of x from main scope

➡ Defined new variable x in this scope

➡ Back to the main scope

```
$ g++ -o SimpleScope SimpleScope.cpp
$ ./SimpleScope
in main before scope, x: 1.2
--> i: 0
in local scope before int, x: 2.2
in local scope after int, x: 4
--> i: 1
in local scope before int, x: 3.2
in local scope after int, x: 4
--> i: 2
in local scope before int, x: 4.2
in local scope after int, x: 4
in main after local scope, x: 4.2
```

Scope of Variables

Based on `examples/02/SimpleScope.cpp`

```
#include <iostream>
using namespace std;

int main() {

    double x = 1.2;

    cout << "in main before scope, x: " << x << endl;

    for(int i=0; i<3; ++i) { // just a local scope
        cout << "--> i: " << i << endl;

        x++;
        cout << "in local scope before int, x: " << x << endl;

        int x = 4;
        cout << "in local scope after int, x: " << x << endl;
    }

    //This will not work: it is outside the scope of i
    // cout << i << endl;
    cout << "in main after local scope, x: " << x << endl;

    return 0;
}
```

[This is terrible coding!]

The scope of a name is the block of program where the name is valid and can be used

- A block is delimited by `{}`
- It can be the body of a function or a simple scope defined by the user with `{}`

➡ Changed value of x from main scope

➡ Defined new variable x in this scope

➡ Back to the main scope

```
$ g++ -o SimpleScope SimpleScope.cpp
$ ./SimpleScope
in main before scope, x: 1.2
--> i: 0
in local scope before int, x: 2.2
in local scope after int, x: 4
--> i: 1
in local scope before int, x: 3.2
in local scope after int, x: 4
--> i: 2
in local scope before int, x: 4.2
in local scope after int, x: 4
in main after local scope, x: 4.2
```

Namespaces

- A `namespace` allows you to group declarations that logically belong together

- They provide an easy way for logical separation of parts of a big and/or complex project

- Basically a “scope” for a group of related declarations

- Example: a `namespace` called `physics` could include units, scalars, vectors, etc., along with functions to calculate mean, standard deviation, scalar product of vectors, etc.

- So far, we have encountered the `std namespace`

Coding Namespaces

Based on `examples/02/TwoNamespaces.cpp`

```
#include <iostream>

namespace physics {
  double mean(double a, double b) {
    return (a+b)/2.;
  }
}

namespace foobar {
  double mean(double a, double b) {
    return (a*a+b*b)/2.;
  }
}



int main() {

   double x = 3, y = 4;

   double z1 = physics::mean(x,y);
   std::cout << "physics::mean(" << x << "," << y << ") = " << z1 << std::endl;

   double z2 = foobar::mean(x,y);
   std::cout << "foobar::mean(" << x << "," << y << ") = " << z2 << std::endl;

   return 0;
}
```

- Access elements of the namespace with `::`
- The `std` namespace is defined in `iostream`
- Notice that we are defining more than one variable in a single line for the first time

```
$ g++ -o TwoNamespaces TwoNamespaces.cpp
$ ./TwoNamespaces
physics::mean(3,4) = 3.5
foobar::mean(3,4) = 12.5
```

Coding Namespaces

Based on `examples/02/TwoNamespaces.cpp`

```
#include <iostream>

namespace physics {
  double mean(double a, double b) {
    return (a+b)/2.;
  }
}

namespace foobar {
  double mean(double a, double b) {
    return (a*a+b*b)/2.;
  }
}



int main() {

   double x = 3, y = 4;

   double z1 = physics::mean(x,y);
   std::cout << "physics::mean(" << x << "," << y << ") = " << z1 << std::endl;

   double z2 = foobar::mean(x,y);
   std::cout << "foobar::mean(" << x << "," << y << ") = " << z2 << std::endl;

   return 0;
}
```

[Use `*0.5` rather than `/2.` to optimize]

- Access elements of the namespace with `::`
- The `std` namespace is defined in `iostream`
- Notice that we are defining more than one variable in a single line for the first time

```
$ g++ -o TwoNamespaces TwoNamespaces.cpp
$ ./TwoNamespaces
physics::mean(3,4) = 3.5
foobar::mean(3,4) = 12.5
```

The using namespace directive

Based on examples/02/TwoNamespaces.cpp

```
#include <iostream>

namespace physics {
  double mean(double a, double b) {
    return (a+b)/2.;
  }
}

namespace foobar {
  double mean(double a, double b) {
    return (a*a+b*b)/2.;
  }
}

using namespace std; // make all names in std namespace available

int main() {

   double x = 3, y = 4;

   double z1 = physics::mean(x,y);
   cout << "physics::mean(" << x << "," << y << ") = " << z1 << endl;

   double z2 = foobar::mean(x,y);
   cout << "foobar::mean(" << x << "," << y << ") = " << z2 << endl;

   return 0;
}
```

- With `using namespace` we can provide default namespaces to look through for unqualified names

- Here the compiler will look through `std` once it fails at finding `cout` and `endl` in the code (Python uses similar concepts and procedures)

Modify, compile, and run `TwoNamespaces` and verify that its behaviour remains unchanged

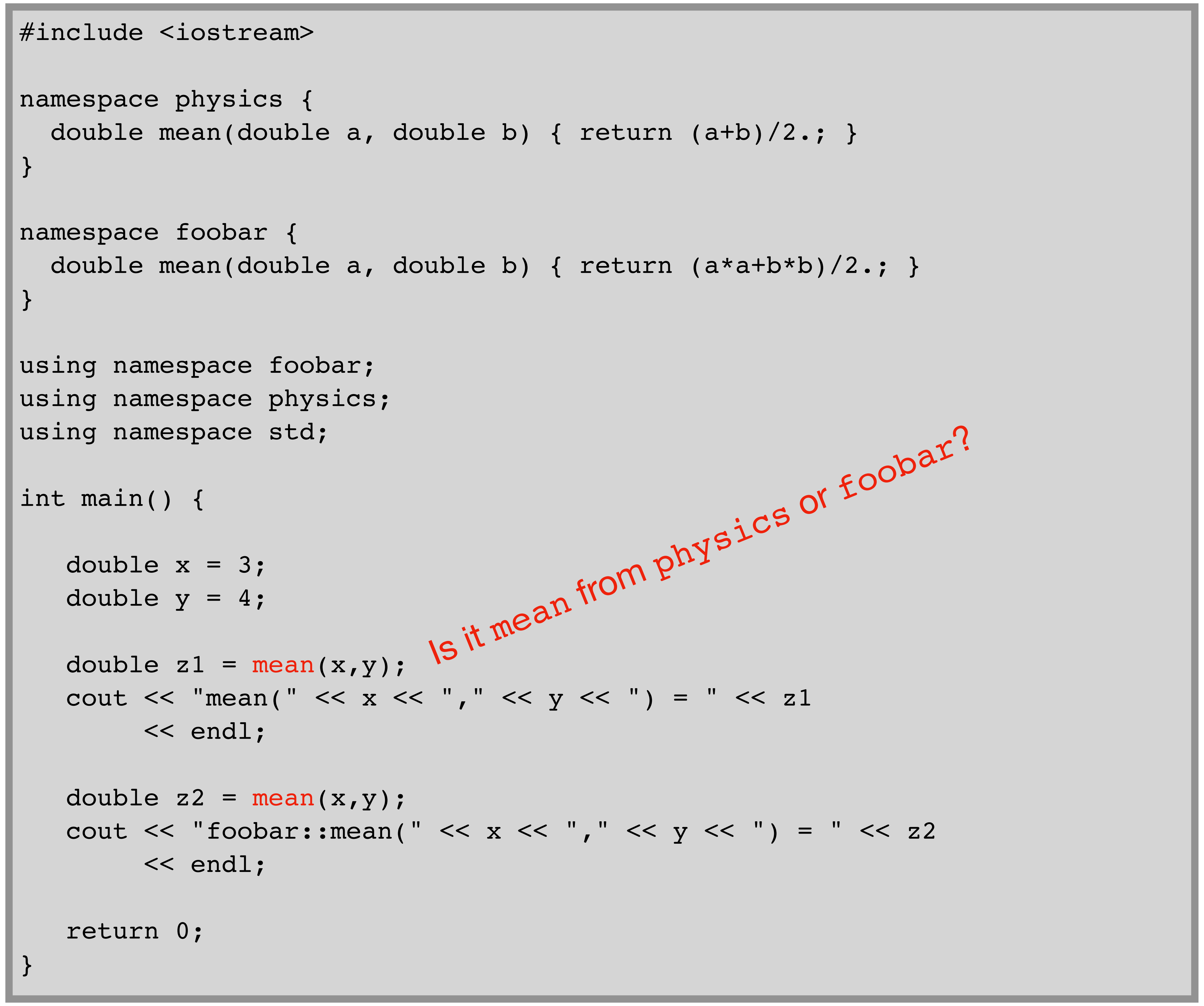
Coding Namespaces

Based on `examples/02/BadNamespaces.cpp`

Some common errors to watch out for

1. Forgetting the namespace when calling an element (e.g., `mean` vs. `physics::mean`) will cause the compiler to not know where to fetch the requested element

2. Using namespaces that have elements with identical names and calling these without the namespace will cause the compiler to not know which element to use out of multiple ones



```
#include <iostream>

namespace physics {
  double mean(double a, double b) { return (a+b)/2.; }
}

namespace foobar {
  double mean(double a, double b) { return (a*a+b*b)/2.; }
}

using namespace foobar;
using namespace physics;
using namespace std;

int main() {

   double x = 3;
   double y = 4;

   double z1 = mean(x,y);
   cout << "mean(" << x << "," << y << ") = " << z1
        << endl;

   double z2 = mean(x,y);
   cout << "foobar::mean(" << x << "," << y << ") = " << z2
        << endl;

   return 0;
}
```

Is it mean from physics or foobar?

Coding Namespaces

Based on `examples/02/BadUsingNamespace.cpp`

```
#include <iostream>

namespace physics {
  double mean(double a, double b) {
    return (a+b)/2.;
  }
}

void printMean(double a, double b) {
  double z1 = physics::mean(a,b);

  using namespace std; // using std namespace within this function!
  cout << "physics::mean(" << a << "," << b << ") = " << z1 << endl;
}

int main() {

  double x = 3;
  double y = 4;

  printMean(x,y);
  cout << "no namespace available in the main!" << endl;

  return 0;
}
```

3. Trying to use a namespace outside its scope

```
$ g++ -o BadUsingNamespace BadUsingNamespace.cpp
$ ./BadUsingNamespace
BadUsingNamespace.cpp:22:3: error: use of undeclared identifier 'cout';
did you mean 'std::cout'?
  cout << "no namespace available in the main!" << endl;
  ^~~~
  std::cout
/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/c++/v1/
iostream:53:33: note: 'std::cout' declared here
extern _LIBCPP_FUNC_VIS ostream cout;
                                ^

BadUsingNamespace.cpp:22:52: error: use of undeclared identifier 'endl';
did you mean 'std::endl'?
  cout << "no namespace available in the main!" << endl;
                                                   ^~~~
                                                   std::endl
/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/c++/v1/
ostream:1004:1: note: 'std::endl' declared here
endl(basic_ostream<_CharT, _Traits>& __os)
^
2 errors generated.
```

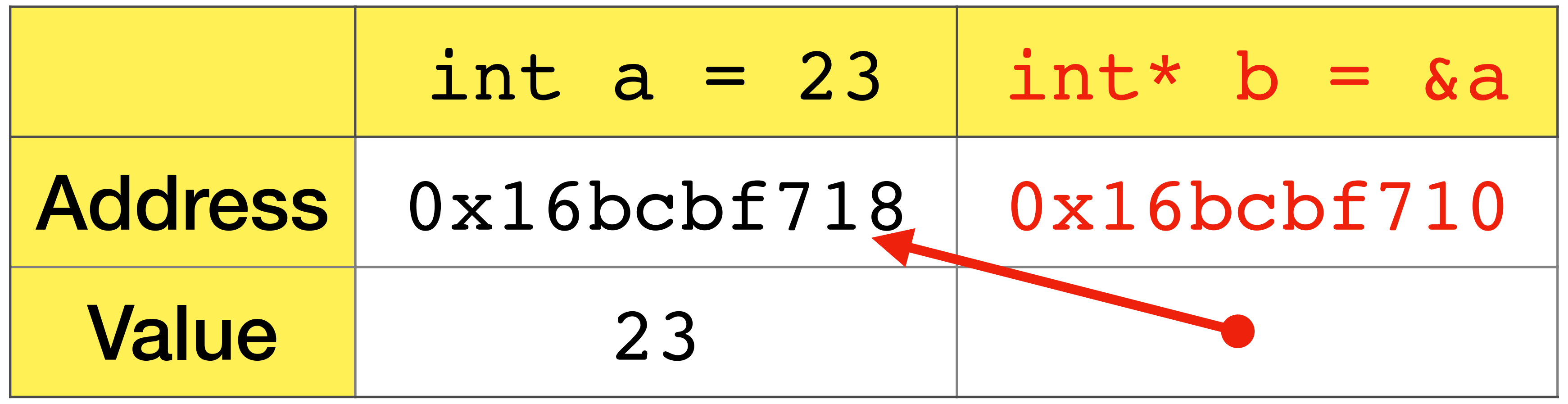
This is a particularly nice compiler:
it suggests solutions!

Pointers and References

	`int a = 23`
Address	`0x16bcbf718`
Value	`23`

- A variable is a label assigned to a location of memory and used by the program to access that location
    - `a` takes up 4 bytes (32 bits) of memory used to store the value 23

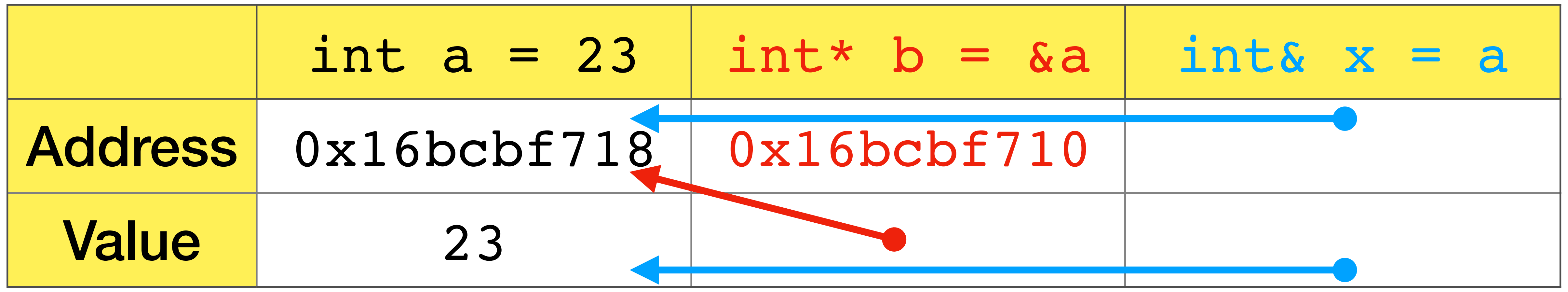
Pointers and References



	`int a = 23`	`int* b = &a`
Address	`0x16bcbf718`	`0x16bcbf710`
Value	`23`	

- A variable is a label assigned to a location of memory and used by the program to access that location
  - `a` takes up 4 bytes (32 bits) of memory used to store the value 23
- `b` is a **pointer** to location of memory named `a`

Pointers and References



	`int a = 23`	`int* b = &a`	`int& x = a`
Address	`0x16bcbf718`	`0x16bcbf710`	
Value	`23`		

- A variable is a label assigned to a location of memory and used by the program to access that location
  - `a` takes up 4 bytes (32 bits) of memory used to store the value 23
- `b` is a **pointer** to location of memory named `a` or `x`
- `x` is a **reference** to `a`
  - It is a different name for the same physical location in memory
  - Using `x` or `a` is exactly the same

Pointers and References

Based on `examples/02/PointersReferences1.cpp`

```
#include <iostream>
using namespace std;

int main() {

   int a;
   cout << "Insert value of a: ";
   cin >> a; // store value provided by user

   int* b; // b is  a pointer to varible of type int
   b  = &a; // value of b is the adress of memory location assigned to a

   int& x = a;

   cout << "value of a: " << a
        << ", address of a, &a: " << &a
        << endl;

   cout << "value of b: " << b
        << ", address of b, &b: " << &b
        << ", value of *b: " << *b
        << endl;

   cout << "value of x: " << x
        << ", address of x, &x: " << &x
        << endl;

   return 0;
}
```

```
$ g++ -o PointersReferences1 PointersReferences1.cpp
$ ./PointersReferences1
Insert value of a: 23
value of a: 23, address of a, &a: 0x16bcbf718
value of b: 0x16bcbf718, address of b, &b: 0x16bcbf710, value of *b: 23
value of x: 23, address of x, &x: 0x16bcbf718
```

Try rerunning and providing a different value: the addresses will not change

Try recompiling and rerunning: the addresses will change

Pointers and References

Based on `examples/02/PointersReferences2.cpp`

```
#include <iostream>
using namespace std;

void print_info(int var, string var_name){
   cout << "value of " << var_name << ": " << var
        << ", address of " << var_name << “, "
        << "&" << var_name << ": " << &var << endl;
}

int main() {

   int a = 1;
   print_info(a, "a");

   int* b = &a;
   *b = 3;
   print_info(a, "a");

   int& x = a;
   x = 45;
   print_info(a, "a");

   return 0;
}
```

- Smarter about printing information by using a (`void`) function
- Change value of `a` by acting on **pointer** `b`
- Change value of `a` by acting on **reference** `x`

```
$ g++ -o PointersReferences2 PointersReferences2.cpp
$ ./PointersReferences2
value of a: 1, address of a, &a: 0x16f5276dc
value of a: 3, address of a, &a: 0x16f5276dc
value of a: 45, address of a, &a: 0x16f5276dc
```

Bad and Null Pointers

Based on `examples/02/BadPointer.cpp`

```
#include <iostream>
using namespace std;

int main() {

   int* b; // b is  a pointer to varible of type int

   int vect[3] = {1,2,3}; // vector of int

   int* c; // non-initialized pointer
           // cout-ing *c may crash at runtime
   cout << "c: " << c << endl;

   for(int i = 0; i<3; ++i) {
     c = &vect[i];
     cout << "c = &vect[" << i << "]: " << c << ", *c: " << *c << endl;
   }

   // bad pointer
   c++;
   cout << "c: " << c << ", *c: " << *c <<endl;

   // null pointer causing trouble
   c = 0;
   cout << "c: " << c << endl;
   cout << "*c: " << *c <<endl;

   return 0;
}
```

- No problem at compile time
- You can try this out 🧠

```
$ g++ -o BadPointer BadPointer.cpp
$ ./BadPointer
c: 0x0
c = &vect[0]: 0x16fdc7728, *c: 1
c = &vect[1]: 0x16fdc772c, *c: 2
c = &vect[2]: 0x16fdc7730, *c: 3
c: 0x16fdc7734, *c: 0
c: 0x0
Segmentation fault: 11
```

Pointers and References in Functions

Based on `examples/02/FuncArgs1.cpp`

- Arguments of functions can be passed in two different ways

1. By value

```
// x is a local variable in f1()
void f1(double x) {
  cout << "f1: input value of x = " << x << endl;
  x = 1.234;
  cout << "f1: change value of x in f1(). x = " << x << endl;
}
```

2. By pointer/reference

```
// x is a reference to argument used by caller
void f2(double& x) {
  cout << "f2: input value of x = " << x << endl;
  x = 1.234;
  cout << "f2: change value of x in f2(). x = " << x << endl;
}
```

Pointers and References in Functions

Based on `examples/02/FuncArgs1.cpp`

- Arguments of functions can be passed in two different ways

1. By value

```
double a = 1;

// This call DOES NOT CHANGE the value of a
// because x contains a local copy of the value of a
f1(a);
```

2. By pointer/reference

```
double a = 1;

// This call CHANGES the value of a to 1.234 (the value of x)
// because x references a directly
f2(a);
```

Pointers and References in Functions

Based on `examples/02/FuncArgs1.cpp`

- Arguments of functions can be passed in two different ways

1. By value

```
double a = 1;

// This call DOES NOT CHANGE the value of a
// because x contains a local copy of the value of a
f1(a);
```

2. By pointer/reference

```
double a = 1;

// This call CHANGES the value of a to 1.234 (the value of x)
// because x references a directly
f2(a);
```

Test out `FuncArgs1.cpp`!

Pointer Arithmetic

Based on `examples/02/PointerArithmetic1.cpp`

```
#include <iostream>

void d_info(int* d){
   using namespace std;
   cout << "d = " << d << ", *d: " << *d <<endl;
}

int main() {

   int v3[] = { 1, 2, 3, 4, 5, 6, 7 }; // array of size 7

   int* d = v3;

   d_info(d++); // post-increment of d
   d_info(d);

   d_info(d+3); // 3 after address d

   d_info(d+7); // 7 after address d

   d_info(d+10); // 10 after address d

   return 0;
}
```

Pointer Arithmetic

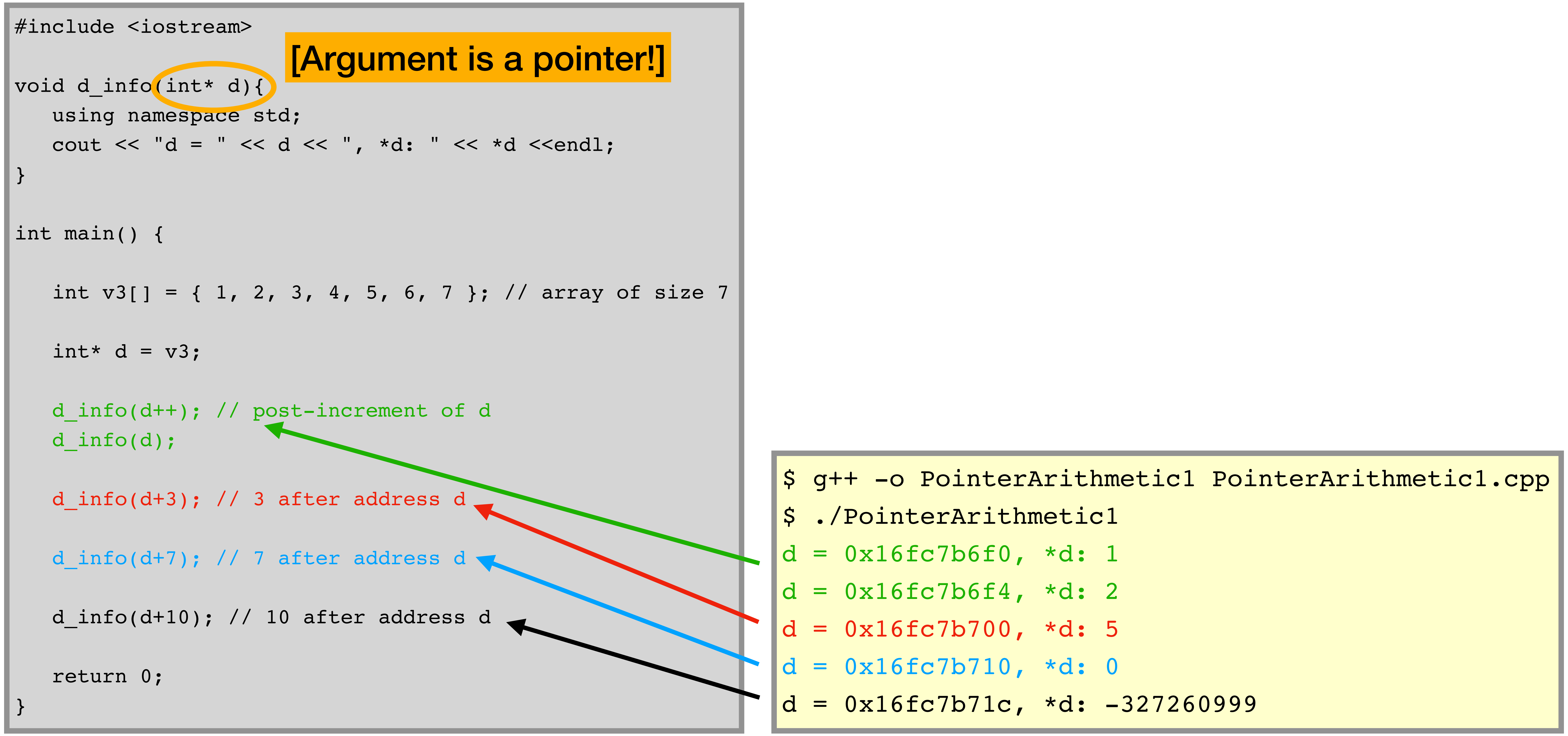
Based on `examples/02/PointerArithmetic1.cpp`

[Argument is a pointer!]

```cpp
#include <iostream>

void d_info(int* d){
   using namespace std;
   cout << "d = " << d << ", *d: " << *d <<endl;
}

int main() {

   int v3[] = { 1, 2, 3, 4, 5, 6, 7 }; // array of size 7

   int* d = v3;

   d_info(d++); // post-increment of d
   d_info(d);

   d_info(d+3); // 3 after address d

   d_info(d+7); // 7 after address d

   d_info(d+10); // 10 after address d

   return 0;
}
```

Pointer Arithmetic

Based on `examples/02/PointerArithmetic1.cpp`



```
#include <iostream>

void d_info(int* d){
   using namespace std;
   cout << "d = " << d << ", *d: " << *d <<endl;
}

int main() {

   int v3[] = { 1, 2, 3, 4, 5, 6, 7 }; // array of size 7

   int* d = v3;

   d_info(d++); // post-increment of d
   d_info(d);

   d_info(d+3); // 3 after address d

   d_info(d+7); // 7 after address d

   d_info(d+10); // 10 after address d

   return 0;
}
```

[Argument is a pointer!]

```
$ g++ -o PointerArithmetic1 PointerArithmetic1.cpp
$ ./PointerArithmetic1
d = 0x16fc7b6f0, *d: 1
d = 0x16fc7b6f4, *d: 2
d = 0x16fc7b700, *d: 5
d = 0x16fc7b710, *d: 0
d = 0x16fc7b71c, *d: -327260999
```

Pointer Arithmetic

Based on `examples/02/PointerArithmetic1.cpp`

```
#include <iostream>

void d_info(int* d){
   using namespace std;
   cout << "d = " << d << ", *d: " << *d <<endl;
}

int main() {

   int v3[] = { 1, 2, 3, 4, 5, 6, 7 }; // array of size 7

   int* d = v3;

   d_info(d++); // post-increment of d
   d_info(d);

   d_info(d+3); // 3 after address d

   d_info(d+7); // 7 after address d

   d_info(d+10); // 10 after address d

   return 0;
}
```

[Argument is a pointer!]

	Address	Value
v3[0]	0x16d8d36f0	1
v3[1]	0x16d8d36f4	2
v3[2]	0x16d8d36f8	3
v3[3]	0x16d8d36fc	4
v3[4]	0x16d8d3700	5
v3[5]	0x16d8d3704	6
v3[6]	0x16d8d3708	7

●—● int *d = v3

```
$ g++ -o PointerArithmetic1 PointerArithmetic1.cpp
$ ./PointerArithmetic1
d = 0x16fc7b6f0, *d: 1
d = 0x16fc7b6f4, *d: 2
d = 0x16fc7b700, *d: 5
d = 0x16fc7b710, *d: 0
d = 0x16fc7b71c, *d: -327260999
```

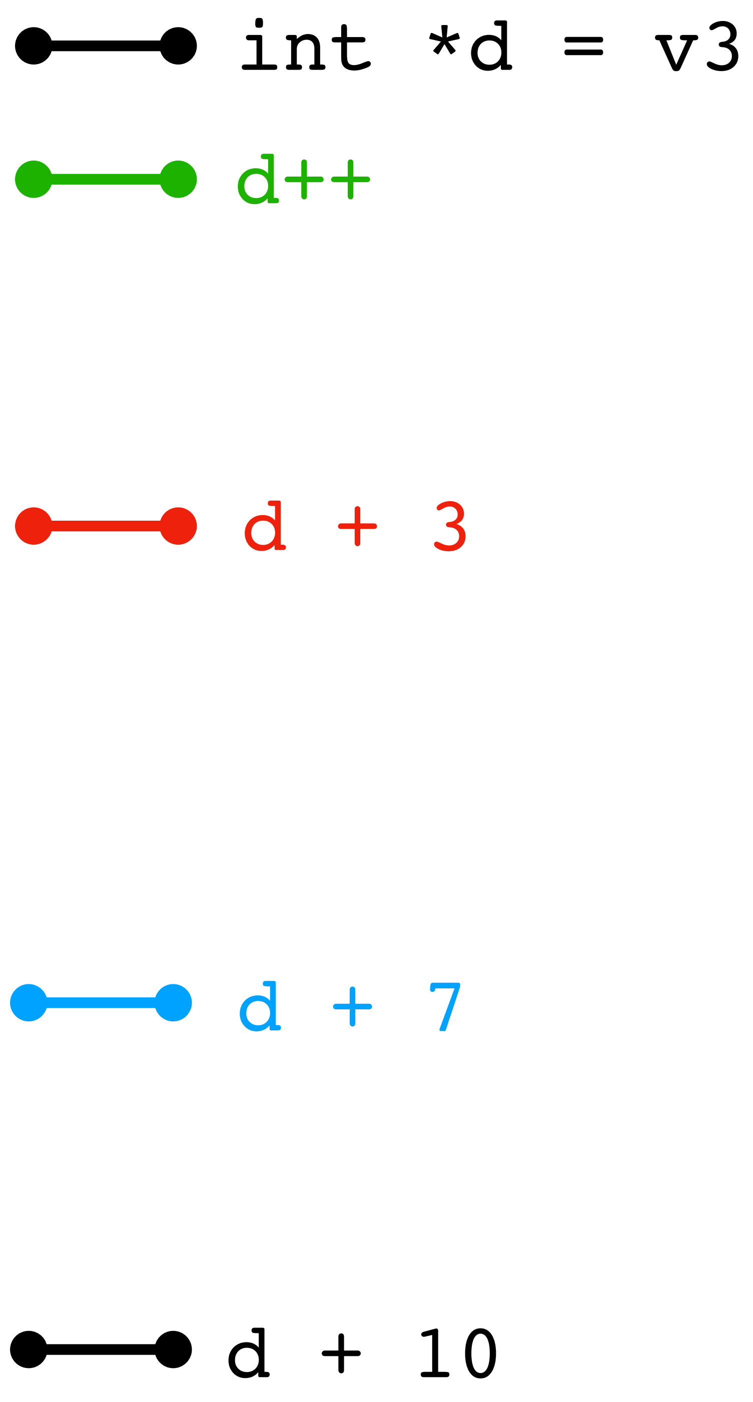
Pointer Arithmetic

Based on `examples/02/PointerArithmetic1.cpp`

```
#include <iostream>

void d_info(int* d){
   using namespace std;
   cout << "d = " << d << ", *d: " << *d <<endl;
}

int main() {

   int v3[] = { 1, 2, 3, 4, 5, 6, 7 }; // array of size 7

   int* d = v3;

   d_info(d++); // post-increment of d
   d_info(d);

   d_info(d+3); // 3 after address d

   d_info(d+7); // 7 after address d

   d_info(d+10); // 10 after address d

   return 0;
}
```

[Argument is a pointer!]

	Address	Value
v3[0]	0x16d8d36f0	1
v3[1]	0x16d8d36f4	2
v3[2]	0x16d8d36f8	3
v3[3]	0x16d8d36fc	4
v3[4]	0x16d8d3700	5
v3[5]	0x16d8d3704	6
v3[6]	0x16d8d3708	7



```
$ g++ -o PointerArithmetic1 PointerArithmetic1.cpp
$ ./PointerArithmetic1
d = 0x16fc7b6f0, *d: 1
d = 0x16fc7b6f4, *d: 2
d = 0x16fc7b700, *d: 5
d = 0x16fc7b710, *d: 0
d = 0x16fc7b71c, *d: -327260999
```

Pointer Arithmetic

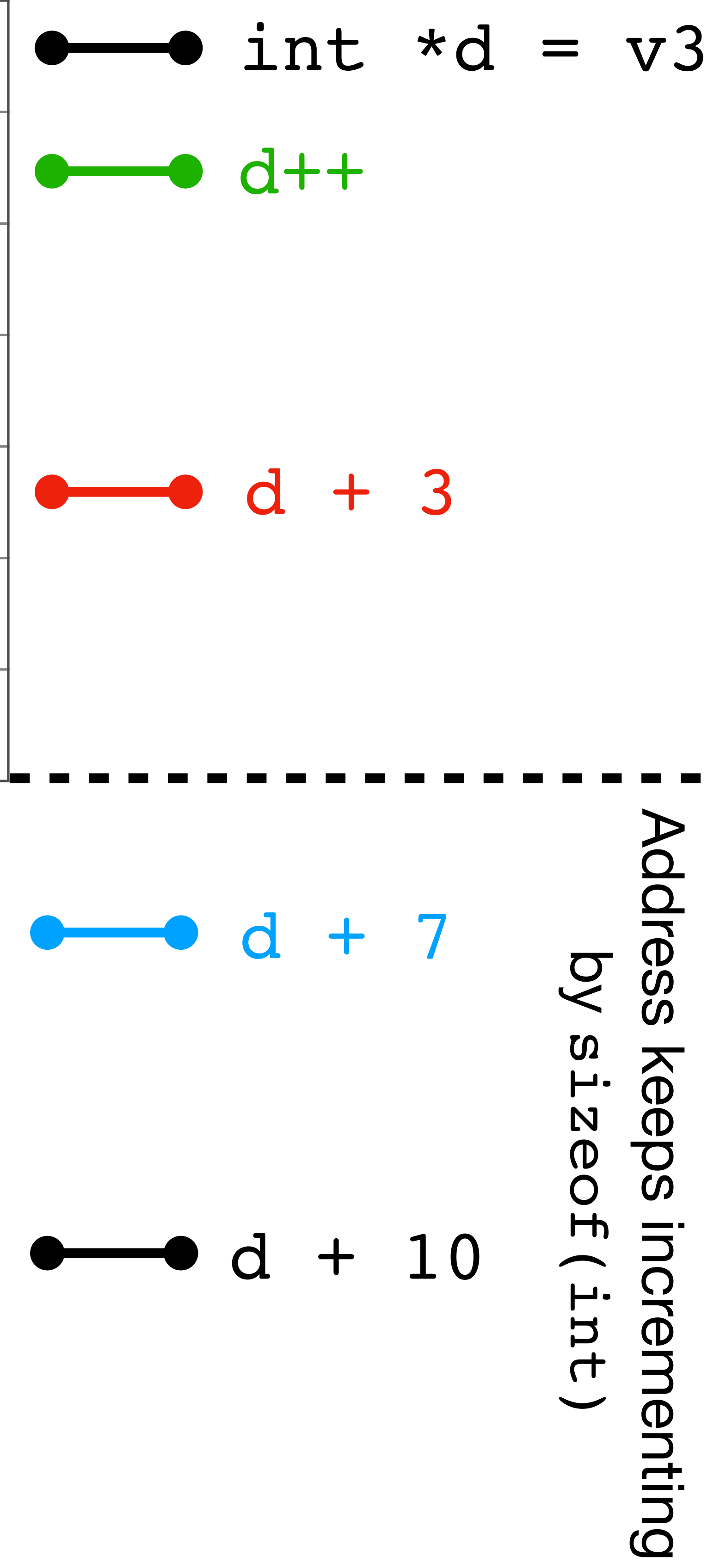
Based on `examples/02/PointerArithmetic1.cpp`

```
#include <iostream>

void d_info(int* d){
   using namespace std;
   cout << "d = " << d << ", *d: " << *d <<endl;
}

int main() {

   int v3[] = { 1, 2, 3, 4, 5, 6, 7 }; // array of size 7

   int* d = v3;

   d_info(d++); // post-increment of d
   d_info(d);

   d_info(d+3); // 3 after address d

   d_info(d+7); // 7 after address d

   d_info(d+10); // 10 after address d

   return 0;
}
```

[Argument is a pointer!]

```
$ g++ -o PointerArithmetic1 PointerArithmetic1.cpp
$ ./PointerArithmetic1
d = 0x16fc7b6f0, *d: 1
d = 0x16fc7b6f4, *d: 2
d = 0x16fc7b700, *d: 5
d = 0x16fc7b710, *d: 0
d = 0x16fc7b71c, *d: -327260999
```

	Address	Value
v3[0]	0x16d8d36f0	1
v3[1]	0x16d8d36f4	2
v3[2]	0x16d8d36f8	3
v3[3]	0x16d8d36fc	4
v3[4]	0x16d8d3700	5
v3[5]	0x16d8d3704	6
v3[6]	0x16d8d3708	7



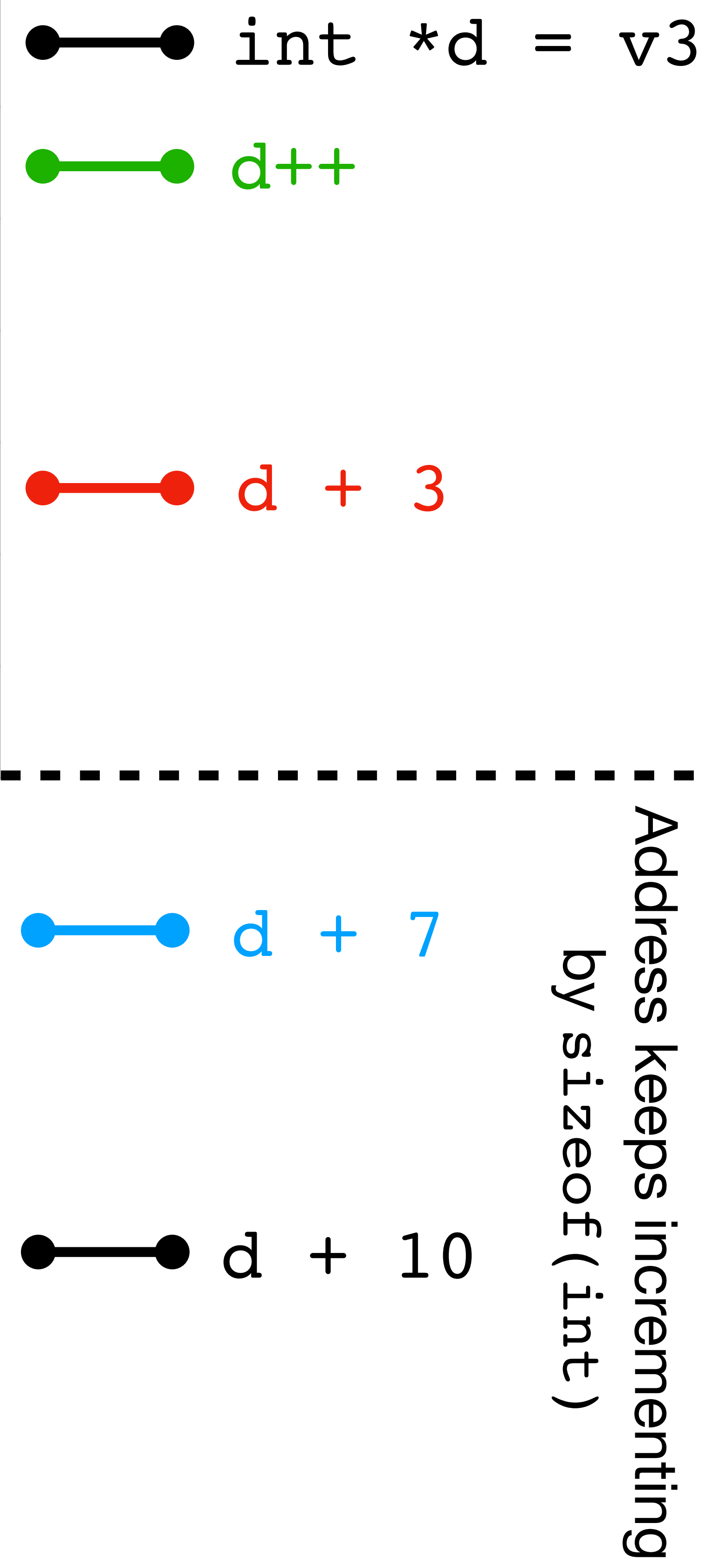
Pointer Arithmetic

Based on `examples/02/PointerArithmetic1.cpp`

```
#include <iostream>

void d_info(int* d){
   using namespace std;
   cout << "d = " << d << ", *d: " << *d <<endl;
}

int main() {

   int v3[] = { 1, 2, 3, 4, 5, 6, 7 }; // array of size 7

   int* d = v3;

   d_info(d++); // post-increment of d
   d_info(d);

   d_info(d+3); // 3 after address d

   d_info(d+7); // 7 after address d

   d_info(d+10); // 10 after address d

   return 0;
}
```

[Argument is a pointer!]

	Address	Value
v3[0]	0x16d8d36f0	1
v3[1]	0x16d8d36f4	2
v3[2]	0x16d8d36f8	3
v3[3]	0x16d8d36fc	4
v3[4]	0x16d8d3700	5
v3[5]	0x16d8d3704	6
v3[6]	0x16d8d3708	7



```
$ g++ -o PointerArithmetic1 PointerArithmetic1.cpp
$ ./PointerArithmetic1
d = 0x16fc7b6f0, *d: 1
d = 0x16fc7b6f4, *d: 2
d = 0x16fc7b700, *d: 5
d = 0x16fc7b710, *d: 0
d = 0x16fc7b71c, *d: -327260999
```

Default initialization over a few extra addresses

Pointer Arithmetic

Based on `examples/02/PointerArithmetic2.cpp`

```cpp
#include <iostream>

using namespace std;

void print_info(int *var, string var_name){
   cout << var_name << " = " << var << ", *" << var_name << ": " << *var <<endl;
}

int main() {

   int v[] = { 10, 20, 300, 40, 50, 60, 70 }; // array of size 7
   int *d = v;
   int *c = &v[4];

   for (int i=0; i<7; i++){
       string var_name = "v[" + to_string(i) + “]";
       print_info(&v[i], var_name);
   }
   print_info(d, "d");
   print_info(c, "c");

   int f = c - d;
   print_info(&f, "c-d");

   f = d - c;
   print_info(&f, "d-c");

   int *h = &v[6] + (d-c);
   print_info(h, "int *h = &v3[6] + (d-c)");

   return 0;
}
```

- Number type to string conversation
- Operating on pointers (memory addresses)

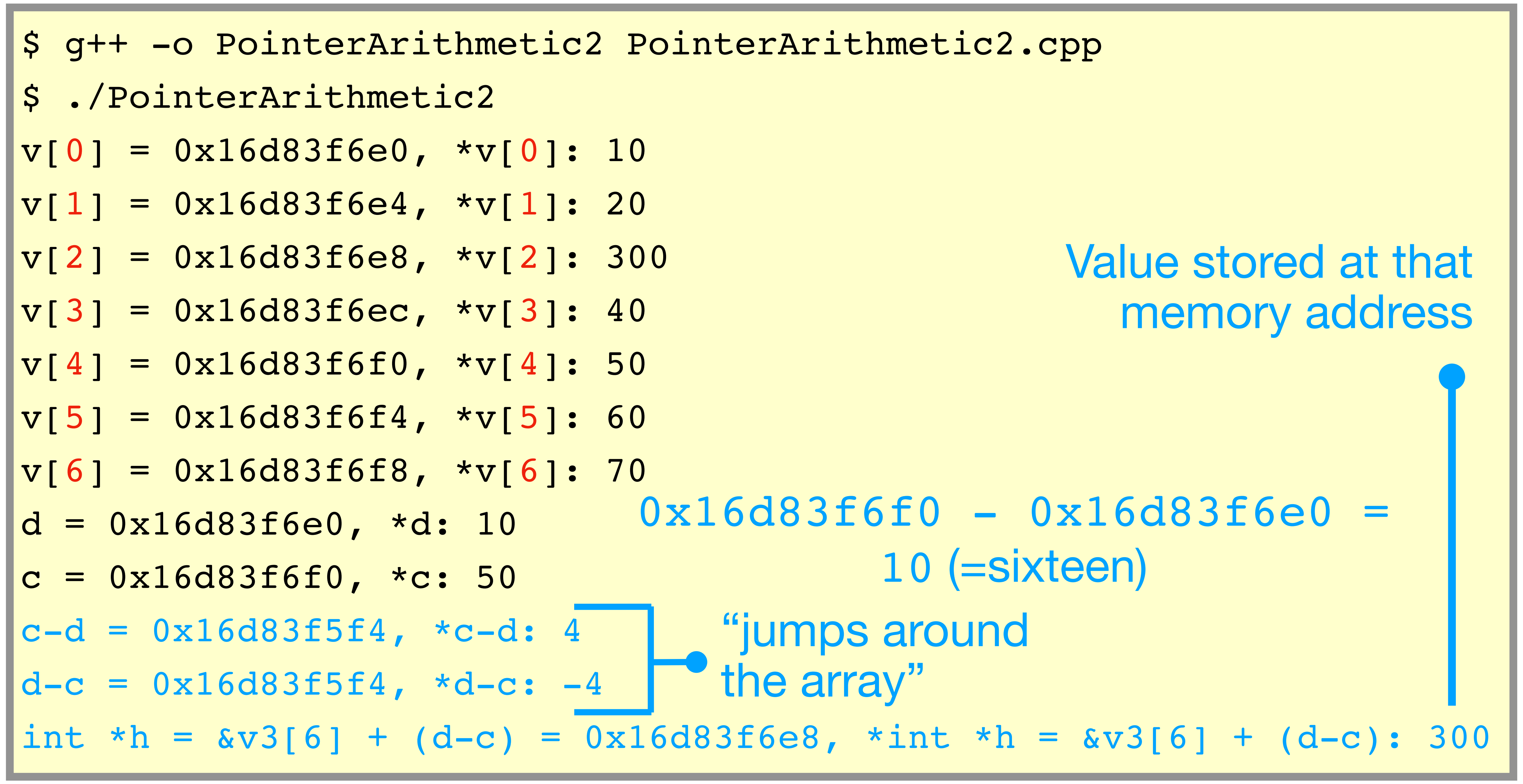
```
$ g++ -o PointerArithmetic2 PointerArithmetic2.cpp
$ ./PointerArithmetic2
v[0] = 0x16d83f6e0, *v[0]: 10
v[1] = 0x16d83f6e4, *v[1]: 20
v[2] = 0x16d83f6e8, *v[2]: 300
v[3] = 0x16d83f6ec, *v[3]: 40
v[4] = 0x16d83f6f0, *v[4]: 50
v[5] = 0x16d83f6f4, *v[5]: 60
v[6] = 0x16d83f6f8, *v[6]: 70
d = 0x16d83f6e0, *d: 10
c = 0x16d83f6f0, *c: 50
c-d = 0x16d83f5f4, *c-d: 4
d-c = 0x16d83f5f4, *d-c: -4
int *h = &v3[6] + (d-c) = 0x16d83f6e8, *int *h = &v3[6] + (d-c): 300
```

Pointer Arithmetic

Based on examples/02/PointerArithmetic2.cpp

```
#include <iostream>

using namespace std;

void print_info(int *var, string var_name){
   cout << var_name << " = " << var << ", *" << var_name << ": " << *var <<endl;
}

int main() {

   int v[] = { 10, 20, 300, 40, 50, 60, 70 }; // array of size 7
   int *d = v;
   int *c = &v[4];

   for (int i=0; i<7; i++){
       string var_name = "v[" + to_string(i) + “]";
       print_info(&v[i], var_name);
   }
   print_info(d, "d");
   print_info(c, "c");

   int f = c - d;
   print_info(&f, "c-d");

   f = d - c;
   print_info(&f, "d-c");

   int *h = &v[6] + (d-c);
   print_info(h, "int *h = &v3[6] + (d-c)");

   return 0;
}
```

- Number type to string conversation
- Operating on pointers (memory addresses)



```
$ g++ -o PointerArithmetic2 PointerArithmetic2.cpp
$ ./PointerArithmetic2
v[0] = 0x16d83f6e0, *v[0]: 10
v[1] = 0x16d83f6e4, *v[1]: 20
v[2] = 0x16d83f6e8, *v[2]: 300
v[3] = 0x16d83f6ec, *v[3]: 40
v[4] = 0x16d83f6f0, *v[4]: 50
v[5] = 0x16d83f6f4, *v[5]: 60
v[6] = 0x16d83f6f8, *v[6]: 70
d = 0x16d83f6e0, *d: 10
c = 0x16d83f6f0, *c: 50
c-d = 0x16d83f5f4, *c-d: 4
d-c = 0x16d83f5f4, *d-c: -4
int *h = &v3[6] + (d-c) = 0x16d83f6e8, *int *h = &v3[6] + (d-c): 300
```

Value stored at that memory address

0x16d83f6f0 – 0x16d83f6e0 = 10 (=sixteen)

“jumps around the array”

Constants

- C++ allows to ensure that the value of a variable, pointer, reference, etc. is unchanged (within its scope)
- Constants **must be initialized**: the developer sets them, not the user!

Constants

- C++ allows to ensure that the value of a variable, pointer, reference, etc. is unchanged (within its scope)
- Constants **must be initialized**: the developer sets them, not the user!

🧠 Try out the code snippets below and witness the compilation errors

Constants

- C++ allows to ensure that the value of a variable, pointer, reference, etc. is unchanged (within its scope)
- Constants **must be initialized**: the developer sets them, not the user!

🧠 Try out the code snippets below and witness the compilation errors

Constant pointer

```
int a = 1, c = 3;

// From right to left: constant pointer to int
int * const b = &a;

// Change value of what b points to: OK!
*b = 5;

// Assign new value (an address) to b: NOT OK!
b = &c;
```

Constants

- C++ allows to ensure that the value of a variable, pointer, reference, etc. is unchanged (within its scope)
- Constants **must be initialized**: the developer sets them, not the user!

Try out the code snippets below and witness the compilation errors

Constant pointer

```
int a = 1, c = 3;

// From right to left: constant pointer to int
int * const b = &a;

// Change value of what b points to: OK!
*b = 5;

// Assign new value (an address) to b: NOT OK!
b = &c;
```

Pointer to constant

```
int a = 1, c = 3;

// From right to left: pointer to constant int
const int * b = &a;

// Change value of what b points to: NOT OK!
*b = 5;

// Assign new value (an address) to b: OK!
b = &c;
```

Constants

Based on `examples/02/BadConstant.cpp`

- A constant pointer to a constant object is the most restrictive case

- The pointer and the value it points to cannot be changed

Constant pointer to constant float

```
int main() {

   float a = 1, c = 3;

   // From right to left: constant pointer to constant float
   const float * const b = &a;

   // Change value of what b points to: NOT OK!
   *b = 5;

   // Assign new value (address) to b: NOT OK!
   b = &c;

   return 0;
}
```

Constants

Based on `examples/02/BadConstant.cpp`

- A constant pointer to a constant object is the most restrictive case

- The pointer and the value it points to cannot be changed

Constant pointer to constant float

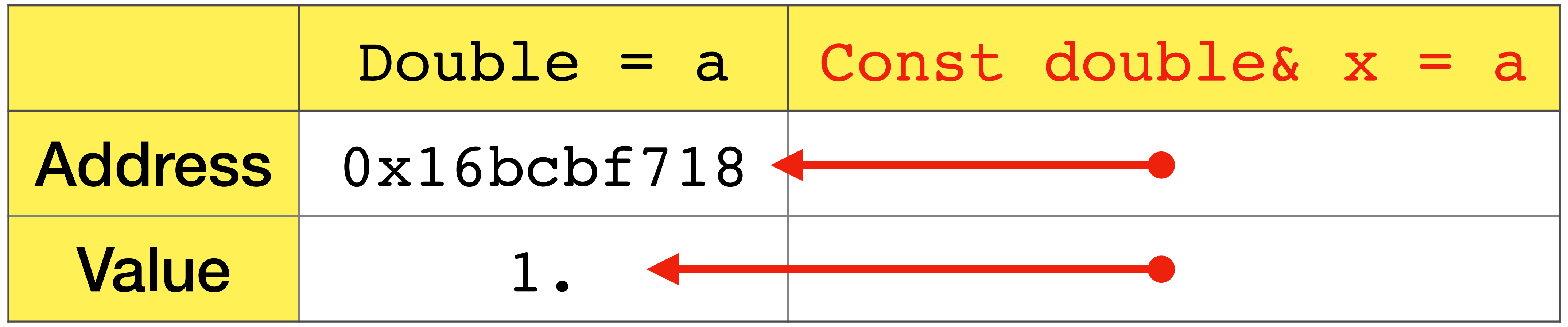
```
int main() {

   float a = 1, c = 3;

   // From right to left: constant pointer to constant float
   const float * const b = &a;

   // Change value of what b points to: NOT OK!
   *b = 5;

   // Assign new value (address) to b: NOT OK!
   b = &c;

   return 0;
}
```

```
$ g++ -o BadConstant BadConstant.cpp
BadConstant.cpp:9:7: error: read-only variable is not assignable
   *b = 5;
   ~~ ^
BadConstant.cpp:12:6: error: cannot assign to variable 'b' with const-qualified type 'const float *const'
   b = &c;
   ~ ^
BadConstant.cpp:6:24: note: variable 'b' declared const here
   const float * const b = &a;
   ~~~~~~~~~~~~~~~~~~~~^~~~~~
2 errors generated.
```

Constant Pointers and References in Functions

Based on `examples/02/FuncArgs2.cpp`

```cpp
#include <iostream>

using namespace std;

void f2(const double& x) {
  cout << "f2: input value of x = "
       << x << endl;
  x = 1.234;
  cout << "f2: change value of x in f2(). x = "
       << x << endl;
}

int main() {

  double a = 1.;

  f2(a);

  return 0;
}
```



	Double = a	Const double& x = a
Address	0x16bcbf718	
Value	1.	

```
$ g++ -o FuncArgs2 FuncArgs2.cpp
$ ./FuncArgs2
FuncArgs2.cpp:8:5: error: cannot assign to variable 'x'
with const-qualified type 'const double &'
  x = 1.234;
  ~ ^
FuncArgs2.cpp:5:23: note: variable 'x' declared const
here
void f2(const double& x) {
        ~~~~~~~~~~~~~~^
1 error generated.
```

Application: Computing Mean and Standard Deviation

Given 7 Numbers, Calculate their Mean

Based on `examples/02/Mean1.cpp`

```
#include <iostream>

using namespace std;

void computeMean(const double* data, int nData, double& mean) {
  mean = 0.;
  for(int i=0; i<nData; ++i) {
    cout << "data: " << data << ", *data: " << *data << endl;
    mean += *data;
    data++;
  }
  mean /= nData; // divide by number of data points
}

int main() {
   double pressure[] = { 1.2, 0.9, 1.34, 1.67, 0.87, 1.04, 0.76 };
   double average;

   computeMean(pressure, 7, average);

   cout << "average pressure: " << average << endl;

   return 0;
}
```

THE STANDARD CHECKLIST

1. Does it compile?
2. Does it run?
3. Is the output correct/understood?

Given 7 Numbers, Calculate their Mean

Based on `examples/02/Mean1.cpp`

```cpp
#include <iostream>

using namespace std;

void computeMean(const double* data, int nData, double& mean) {
  mean = 0.;
  for(int i=0; i<nData; ++i) {
    cout << "data: " << data << ", *data: " << *data << endl;
    mean += *data;
    data++;
  }
  mean /= nData; // divide by number of data points
}

int main() {
   double pressure[] = { 1.2, 0.9, 1.34, 1.67, 0.87, 1.04, 0.76 };
   double average;

   computeMean(pressure, 7, average);

   cout << "average pressure: " << average << endl;

   return 0;
}
```

THE STANDARD CHECKLIST

1. Does it compile?
2. Does it run?
3. Is the output correct/understood?

```
$ g++ -o Mean1 Mean1.cpp
$ ./Mean1
data: 0x16d96f710, *data: 1.2
data: 0x16d96f718, *data: 0.9
data: 0x16d96f720, *data: 1.34
data: 0x16d96f728, *data: 1.67
data: 0x16d96f730, *data: 0.87
data: 0x16d96f738, *data: 1.04
data: 0x16d96f740, *data: 0.76
average pressure: 1.11143
```

Given 7 Numbers, Calculate their Mean

Based on `examples/02/Mean1.cpp`

```
#include <iostream>

using namespace std;

void computeMean(const double* data, int nData, double& mean) {
  mean = 0.;
  for(int i=0; i<nData; ++i) {
    cout << "data: " << data << ", *data: " << *data << endl;
    mean += *data;
    data++;
  }
  mean /= nData; // divide by number of data points
}

int main() {
   double pressure[] = { 1.2, 0.9, 1.34, 1.67, 0.87, 1.04, 0.76 };
   double average;

   computeMean(pressure, 7, average);

   cout << "average pressure: " << average << endl;

   return 0;
}
```

Critical review of the code

- Input data passed as constant pointer
  - Good: cannot cause trouble to caller! Data integrity is guaranteed
- Number of data points passed by value
  - Simple `int`: no gain in passing by reference
  - Bad: separate variable from array of data, exposes to user error
- Function design
  - Very bad: `void` function with no return type
  - Good: appropriate name. computeMean() suggests an action not a type

Implementation with Return Type

Based on `examples/02/Mean2.cpp`

```
#include <iostream>

using namespace std;

double mean(const double* data, int nData) {
  double mean = 0.;
  for(int i=0; i<nData; ++i) {
    cout << "data: " << data << ", *data: " << *data << endl;
    mean += *data;
    data++;
  }
  mean /= nData; // divide by number of data points
  return mean;
}

int main() {
   double pressure[] = { 1.2, 0.9, 1.34, 1.67, 0.87, 1.04, 0.76 };
   double average = mean(pressure, 8);

   cout << "average pressure: " << average << endl;

   return 0;
}
```

- Mostly the same function but we make it return the mean
- New name to make it explicit function returns something: not a rule, but courtesy to code viewers/users
- Still exposed to user error...

Implementation with Return Type

Based on `examples/02/Mean2.cpp`

```cpp
#include <iostream>

using namespace std;

double mean(const double* data, int nData) {
  double mean = 0.;
  for(int i=0; i<nData; ++i) {
    cout << "data: " << data << ", *data: " << *data << endl;
    mean += *data;
    data++;
  }
  mean /= nData; // divide by number of data points
  return mean;
}

int main() {
   double pressure[] = { 1.2, 0.9, 1.34, 1.67, 0.87, 1.04, 0.76 };
   double average = mean(pressure, 8);

   cout << "average pressure: " << average << endl;

   return 0;
}
```

- Mostly the same function but we make it return the mean
- New name to make it explicit function returns something: not a rule, but courtesy to code viewers/users
- Still exposed to user error...

```
$ g++ -o Mean2 Mean2.cpp
$ ./Mean2
data: 0x16d24b710, *data: 1.2
data: 0x16d24b718, *data: 0.9
data: 0x16d24b720, *data: 1.34
data: 0x16d24b728, *data: 1.67
data: 0x16d24b730, *data: 0.87
data: 0x16d24b738, *data: 1.04
data: 0x16d24b740, *data: 0.76
data: 0x16d24b748, *data: 1.56953e-282
average pressure: 0.9725
```

[The mean is wrong!]

Compute Mean and Standard Deviation of Data

```
void computeMean(const double* data, int nData, double& mean, double& stdDev) {
 // Two variables passed by reference to void function: not great, but harmless
}

double meanWithStdDev(const double* data, int nData, double& stdDev) {
  // Error passed by reference to mean function! Ugly and anti-intuitive!
}

double mean(const double* data, int nData) {
  // One method to compute only average
}

double stdDev(const double* data, int nData) {
  // One method to compute standard deviation
  // Use mean() to compute average needed by std deviation
}
```

Compute Mean and Standard Deviation of Data

Solution in `examples/02/Mean3.cpp`

```
void computeMean(const double* data, int nData, double& mean, double& stdDev) {
 // Two variables passed by reference to void function: not great, but harmless
}

double meanWithStdDev(const double* data, int nData, double& stdDev) {
  // Error passed by reference to mean function! Ugly and anti-intuitive!
}

double mean(const double* data, int nData) {
  // One method to compute only average
}

double stdDev(const double* data, int nData) {
  // One method to compute standard deviation
  // Use mean() to compute average needed by std deviation
}
```

Can you do this?