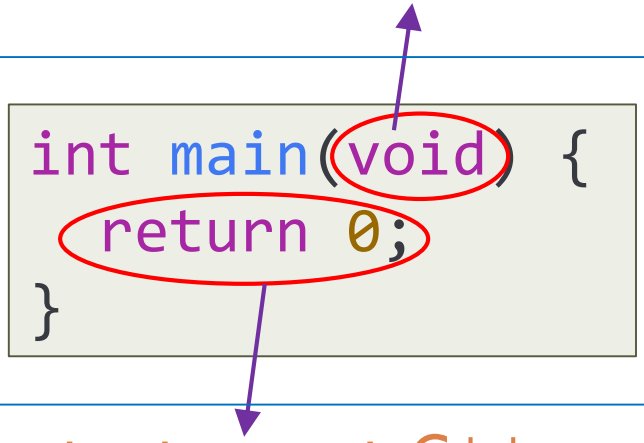
HIGH-LEVEL PROGRAMMING 2

First C++ Programs

by Prasanna Ghali

Simplest C++ Program

In C++, you can specify function that takes no parameters using keyword `void` in parameter list or by using an empty parameter list



If `return` statement not present, C++ compiler will insert `return 0;`

Simplest C++ Program

- I like typing less, so my examples will look like this:

```
int main() {
}
```

Using C Standard Library

- For compatibility with C standard library, C++ standard library provides headers just as in C source files

Just as in C code, C standard library names are also in global scope in C++ code

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  int *pi = (int*) ::malloc(sizeof(int));
  *pi = 39;
  ::printf("*pi: %d\n", *pi);
}
```

Using C Standard Library

- For compatibility with C standard library, C++ standard library provides headers just as in C source files

Don't do this!!!
Will be deprecated in future versions!!!

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  int *pi = (int*) malloc(sizeof(int));
  *pi = 39;
  printf("*pi: %d\n", *pi);
}
```

Using C Standard Library

- Facilities of C standard library in header *name.h* provided in C++ standard library header *cname*:

Names in these headers are within namespace `std`

```
#include <cstdio>
#include <cstdlib>

int main() {
  int *pi = (int*) std::malloc(sizeof(int));
  *pi = 39;
  std::printf("*pi: %d\n", *pi);
}
```

C Standard Library and HLP2 Assessments

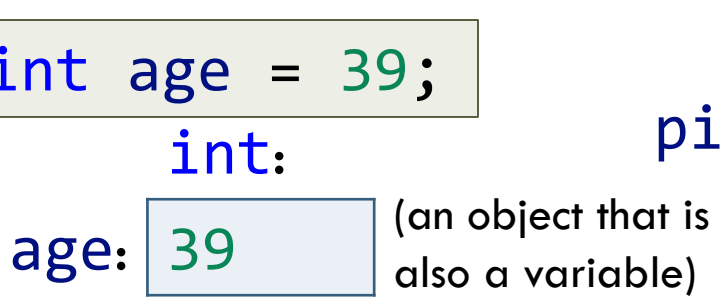
- Unless explicitly specified by assessment specification, expect zero grade for assessments that rely on C standard library for I/O, dynamic memory allocation/deallocation, …

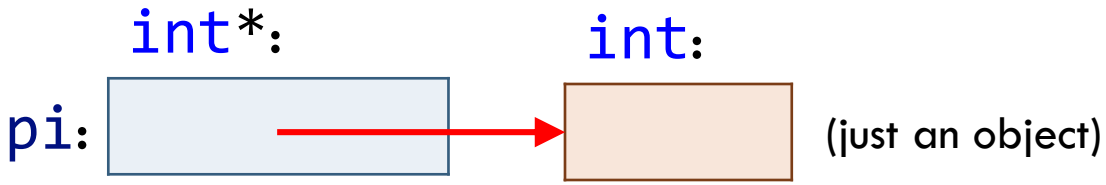
Objects, Variables, Types …

- *Type* is set of values and set of operations on those values
- *Object* is region of memory that has a type
  - So we know what kind of information can be placed in that object
- *Variable* is a named object

```
int *pi = (int*) malloc(sizeof(int));
```

```
int age = 39;
```





Writing To Standard Output

Standard header declares global variables that control reading from and writing to standard streams `stdout`, `stdin`, and `stderr`

```
#include <iostream>

int main() {
  std::cout << "Hello World\n";
}
```

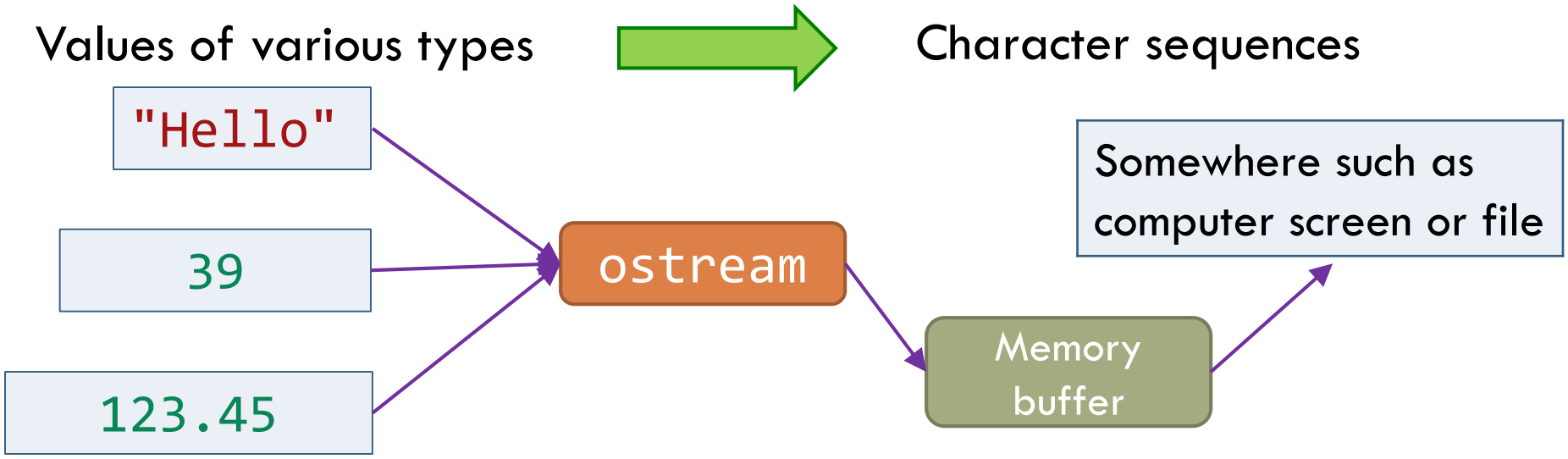
Namespaces are C++ mechanisms that introduce new scopes to avoid conflicts between names in large programs.
`std` is namespace for virtually all names in C++ standard library

`::` is *scope resolution operator*.
`std::cout` means "name `cout` in namespace scope `std`"

Global variable of type `std::ostream` is instantiated at program startup and its purpose is to write characters to standard stream `stdout`

Writing To Standard Output

- `std::ostream` [defined in `<ostream>`] is a type that converts objects into stream [that is, sequence] of characters [that is, bytes]
- `std::cout` is global variable of type `std::ostream` that exclusively writes to output stream `stdout`



Writing To Standard Output

Class `std::ostream` provides member function *overloads* of binary left shift operator for built-in types [`int`, `long`, `float`, `double`, …].
Equivalent to: `(std::cout).operator<<(10);`

```
#include <iostream>

int main() {
  std::cout << 3+7;
  std::cout << "Hello World\n";
}
```

Class `std::ostream` provides non-member function *overloads* of binary left shift operator for inserting characters [`char`, `unsigned char`, `char const*`, …].
Equivalent to: `std::operator<<(std::cout, "Hello World\n");`

Writing To Standard Output

Expression equivalent to: `(std::cout).operator<<(10)`
and it evaluates to `std::cout`

Binary left shift operator `<<` is *left-associative*

```
#include <iostream>

int main() {
  std::cout << 3+7 << "\n" ;
}
```

Expression equivalent to: `std::cout << "\n"`
and it evaluates to `std::operator<<(std::cout, "\n")`

Writing To Standard Output


```
#include <iostream>

int main() {
 std::cout << 3+7 << "\n" ;
}
```

Expression equivalent to:

```
std::operator<<( (std::cout).operator<<(10), "\n" )
```

Writing To Standard Output


```
#include <iostream>

int main() {
  std::cout << "Hello World\n";
  std::cout << "Hello World" << std::endl;
}
```

`std::endl` is output manipulator.

Manipulators are helper functions that change the way a stream formats characters.

Here `std::endl` does two things to `stdout` – the stream to which `std::cout` is writing characters:

1) Outputs newline character `'\n'`
2) Flushes output stream `stdout`

Writing To Standard Output


```
#include <iostream>

int main() {
  std::cout << "Hello World\n";
  std::cout << 3+7 << "\n";
  std::cout << 3.1*7.2 << std::endl;
}
```

```
( (std::cout).operator<<(22.32) ).operator<<(std::endl);
```

Reading From Standard Input

Global variable of type `std::istream` instantiated at program startup to write characters to standard stream `stdin`

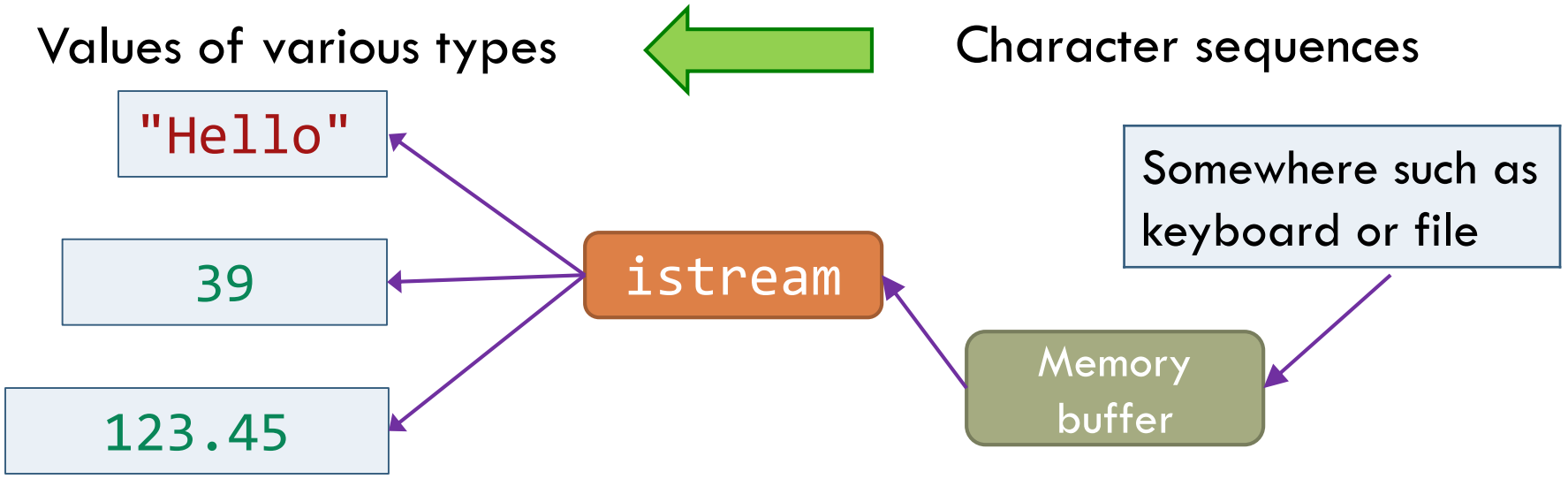
```
#include <iostream>

int main() {
  std::cout << "Enter your first name: ";
  char name[81];
  std::cin >> name;
  std::cout << "Hello " << name << '\n';
}
```

Class `std::istream` provides non-member function *overloads* of binary right shift operator for extracting characters [`char`, `unsigned char`, `char const*`, …].

Equivalent to: `std::operator>>(std::cin, name);`

Reading From Standard Input

- `std::istream` [defined in `<istream>`] is a type that converts stream [that is, sequence] of characters [that is, bytes] to typed objects
- `std::cin` is global variable of type `std::istream` that exclusively reads from input stream `stdin`



Reading From Standard Input


```
#include <iostream>

int main() {
  std::cout << "Enter your first name and age: ";
  char name[81];
  std::cin >> name;
  int age;
  std::cin >> age;
  std::cout << "Hello " << name << " age " << age << "\n";
}
```

Class `std::istream` provides member function *overloads* of binary right shift operator for built-in types [`int`, `long`, `float`, `double`, …].
Equivalent to: `(std::cin).operator>>(age);`

Reading From Standard Input

Expression equivalent to: `std::operator>>(std::cin, name)`
and it evaluates to `std::cin`

```
#include <iostream>

int main() {
  std::cout << "Enter your first name and age: ";
  char name[81];
  int age;
  std::cin >> name >> age;
  std::cout << "Hello " << name << " age " << age << "\n";
}
```

Binary right shift operator << is *left-associative*

Expression equivalent to: `(std::cin).operator>>(age)`
and it evaluates to `std::cin`

Reading From Standard Input


```
#include <iostream>

int main() {
  std::cout << "Enter your first name and age: ";
  char name[81];
  int age;
  std::cin >> name >> age;
  std::cout << "Hello " << name << " age " << age << "\n";
}
```

Expression equivalent to:

```
std::operator>>(std::cin, name).operator>>(age)
```

Ensuring Validity of Input Data

- Surprisingly hard to write robust program that can survive incorrect or wrong data entered by untrained or careless users!!!

Input: Error Handling

- What happens when following program is executed?

```
// input-error01.cpp
std::cout << "Enter integer value: ";
int i;
std::cin >> i; // Enter .75

std::cout << "Enter fractional value: ";
double d = 1.1;
std::cin >> d; // Enter 123.45
std::cout << "i==" << i << " d==" << d << '\n';
```

Since 1st character encountered by `cin` is `'.'` [which cannot be part of any `int` value], `cin` enters a *failed* state!!!
Therefore, subsequent expression `std::cin >> d` has no effect!!!

std::ostream States

- An std::ostream can be in one of four states:

Stream states	
good()	Operations succeeded
eof()	We hit end of input [that is, end of file]
fail()	Something unexpected happened [e.g., looking for a digit character and found '.']
bad()	Something unexpected and serious happened [e.g., a disk read error]

```
std::cout << "Enter integer value: ";
int i;
std::cin >> i; // Enter .75
if (std::cin.fail()) {
  std::cout << "Bad input.\n";
  // Exit program? How to get correct input?
}
```

Input: Error Handling

- Both `istream` and `ostream` inherit a function to clear stream's error state and retry read or write operation …

```
std::cout << "Enter integer value: ";
int i;
std::cin >> i; // user enters .75
if (std::cin.fail()) {
  std::cin.clear(); // clear error state
  std::cin >> i;    // retry again ...
}
std::cout << "i==" << i << "\n";
```

Retry won't work after clearing input stream's error state since characters `'.','7','5'` are still in input stream!!!

We must tell `std::cin` to ignore these characters using inherited `ignore()` function.

Input: Error Handling


```
std::cout << "Enter integer value: ";
int i;
std::cin >> i; // user enters .75
if (std::cin.fail()) {
  std::cin.clear(); // clear error state
  std::cin.ignore(1000, '\n');
  std::cin >> i;    // retry again ...
}
std::cout << "i==" << i << "\n";
```

`std::cin` will ignore either first `1000` characters in stream or all character up to and including delimiting character `'\n'` – whichever occurs first.

Input: Error Handling


```
// input-error02.cpp: more robust version ...
int i;
// as long as input stream cannot read an integer value,
// continue prompting user to provide integer value ...
do {
  if (std::cin.fail()) {
    std::cin.clear(); // clear stream's error state
    std::cin.ignore(1000, '\n'); // ignore characters ...
  }
  std::cout << "Enter integer value: ";
  std::cin >> i;
} while (!std::cin.good());

// ok: finally, we've valid integer data ...
std::cout << "i==" << i << "\n";
```

Anything Wrong Here?

- What happens when following program is executed?

```
// input-error03.cpp:
int main() {
  char str[10];
  int i;
  std::cin >> str >> i;
  // user enters: Supercalifragilistic
  std::cout << "You entered: " << str << " | " << i << "\n";
}
```

Since `sizeof("Supercalifragilistic") > sizeof(str)`, characters typed by user will overflow static array `str` causing stack to be smashed!!!

One solution is to limit program to read only first 9 characters …

Setting Field Width for Input

- Both `istream` and `ostream` inherit a function to manage maximum number of characters read from or written to stream

```
// incorrect version
char str[10];
std::streamsize old_width = std::cin.width(sizeof(str));
std::cin >> str; // user enters: Supercalifragilistic
std::cin.width(old_width);

int i;
std::cin >> i;
std::cout << "You entered: " << str << " | " << i << "\n";
```

Makes `cin` read maximum of 9 characters

Reads first 9 characters `Supercali` but remaining characters `fragilistic` will be read in expression `std::cin >> i` and will cause stream to be in fail state!!! We must tell `std::cin` to ignore characters `fragilistic` using inherited `ignore()` function.

Setting Field Width for Input


```
// input-error04.cpp: more robust version ...
char str[10];
// read maximum 9 characters ...
std::streamsize old_width = std::cin.width(sizeof(str));
std::cin >> str; // user enters: Supercalifragilistic

std::cin.width(old_width);
// ignore any other characters in stream including '\n'
std::cin.ignore(1000, '\n');

int i;
std::cin >> i;
std::cout << "You entered: " << str << " | " << i << "\n";
```

Idiomatic Testing of Streams

- `ostream` or `istream` variable can be used as condition by calling this member function
- In that case, condition is `true` (succeeds) or `false` (fails) if stream's state is `good()` or `!good()`, respectively

```
int i;
do {
  if (std::cin.fail()) {
    std::cin.clear(); // clear stream's error state
    std::cin.ignore(1000, '\n'); // ignore characters ...
  }
  std::cout << "Enter integer value: ";
  std::cin >> i;
} while (!std::cin);

// ok: finally, we've valid integer data ...
std::cout << "i==" << i << "\n";
```

means "`while std::cin` is not good"

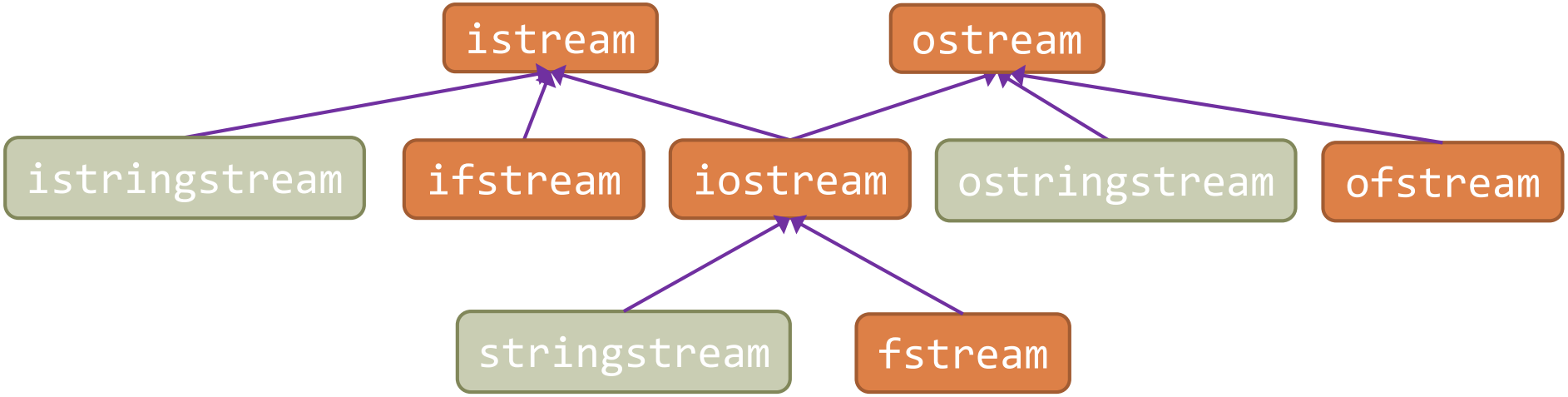
Idiomatic C++ Input Loops

- Example that detects adjacent repeated positive `int` values in sequence of values

```
int previous = -1, current;
// read unknown number of +ve integer values from stdin ...
// signal EOF in Linux using CTRL-D ...
while (std::cin >> current) {
  if (current < 0) continue;
  if (previous == current) {
    std::cout << "repeated value: “ << current << '\n';
  }
  previous = current;
}
```

I/O Streams Hierarchy

- `std::istream` can be connected to input device [e.g., keyboard], file, or `std::string` [a C++ standard library type]
- `std::ostream` can be connected to output device [console window], file, or `std::string`



File Streams

- File stream can be opened either by constructor or by an open( ) call:

Opening files with file stream	
`std::fstream fs;`	Make a file stream variable for opening later
`fs.open(s, m);`	Open a file called `s` [C-style string] with mode `m` and have variable [defined in previous row] `fs` refer to it
`std::fstream fs(s, m);`	Open a file called `s` [C-style string] with mode `m` and make a file stream `fs` refer to it
`fs.is_open();`	Is file referenced by file stream `fs` open?
`fs.close();`	Close file referenced by file stream `fs`

File Streams

- You can open a file in one of several modes:

Opening files with file stream	
`std::ios_base::in`	Open file for reading
`std::ios_base::out`	Open file for writing
`std::ios_base::app`	Open file for appending [i.e., add from end of the file]
`std::ios_base::binary`	Open file so that operations are performed in binary [as opposed to text]
`std::ios_base::ate`	"at end [of file]" [open and seek to the end]
`std::ios_base::trunc`	Truncate file to zero length

File I/O

- File for reading is attached to `istream`
- File for writing is attached to `ostream`
- Since we know how to read from `istream` and write to `ostream`, anything you could do to output stream `stdout` and input stream `stdin`, you can do to files too …
- See *fileio.cpp*

C++ is Strongly Typed

- If C compiler sees undeclared function, it assumes function takes unknown number of parameters and returns an `int`

```
#include <stdio.h>

int main(void) {
  // ok: compiles with C compiler ...
  printf("3+7 == %d\n", add(3, 7));
  return 0;
}

int add(int lhs, int rhs) {
  return lhs+rhs;
}
```

C++ is Strongly Typed

- Unlike C, C++ requires all names be declared before their first use

```
#include <iostream>

int main() {
  // error: call to undeclared function add ...
  std::cout << "3+7 == " << add(3, 7) << "\n";
}

int add(int lhs, int rhs) {
  return lhs+rhs;
}
```

C++ is Strongly Typed

- Unlike C, C++ requires all names be declared before their first use

```
#include <iostream>

int add(int, int);

int main() {
  // ok: function add is now declared ...
  std::cout << "3+7 == " << add(3, 7) << "\n";
}

int add(int lhs, int rhs) {
  return lhs+rhs;
}
```

C++ Integer Types

- Microsoft compiler is 32-bit compiler while GCC and Clang are 64-bit compiler
- `sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`

Type Name	Number of Bytes
`char`	1
`short`	2
`int`	4
`long`	4/8
`long long`	8
`size_t` (declared in `<cstddef>`)	4/8

C++ Boolean Type

- Microsoft compiler is 32-bit compiler while GCC and Clang are 64-bit compiler

Type Name	Number of Bytes	Values
bool	1	true/false

```
int i = 10;
bool b = true;
std::cout << "b: " << b << " | "
          << std::boolalpha << b << "\n";
std::cout << "(i+b): " << (i+b) << "\n";
```

C++ Integer Literals

Literal	Type
`123`	`int`
`0173`	`int`
`0x7b`	`int`
`0b0111'1011`	`int`
`123u`	`unsigned int`
`123l` or `123L`	`long int`
`123u` or `123U`	`unsigned int`
`123ul` or `123UL`	`unsigned long int`
`123ll` or `123LL`	`long long int`
`123ull` or `123ULL`	`unsigned long long int`

C++ Floating-Point Types

- Microsoft compiler is 32-bit compiler while GCC and Clang are 64-bit compiler

Type Name	Number of Bytes
`float`	4
`double`	8
`long double`	8/16

C++ Floating-Point Literals

Literal	Type
`123.`	`double`
`123.f` or `123.f`	`float`
`123.l` or `123.L`	`long long double`
`1.23e2`	`double`
`3.141'592'653'590`	`double`

Querying Properties of Arithmetic Types

- C++ standard library provides way to query properties of arithmetic types

```
#include <iostream>
#include <limits>

std::cout << "min int: " << std::numeric_limits<int>::min() << "\n";
std::cout << “max int: " << std::numeric_limits<int>::max() << "\n";

std::cout << "lowest double: “
          << std::numeric_limits<double>::lowest() << "\n";
std::cout << "min double:    "
          << std::numeric_limits<double>::min() << "\n";
std::cout << "max double:    “
          << std::numeric_limits<double>::max() << "\n";
```

Implicit Conversions

- C++ allows for implicit conversions (to be compatible with C)

```
int a = 20'000;
char c = a; // squeeze large int into small char
int b = c;
if (a != b)
  std::cout << "oops!: " << a << " != " << b << '\n';
else
  std::cout << "Wow!!! C++ has large characters\n";
```

Implicit Conversions

- *Narrowing* conversions are unsafe!!!

```
double d = 0.;
while (std::cin >> d) {
  int i = d;
  char c = i;
  int i2 = c;
  std::cout << "d==" << d  // original double
    << " i==" << i         // converted to int
    << " i2==" << i2       // int value of char
    << " char(" << c << ")\n"; // the char
}
```

Traditional C++ Initialization Syntax


```
// traditional C++ initialization syntaxes
int a = 39;
int b(39);
int c = int(39);
int d = int(); // d initialized to zero

// unsafe initialization
double e = 2.7;
int f(e);
short g = f;
```

Universal and Uniform Initialization

- C++11 introduced new initialization notation to
  - Provide universal way of initializing
  - Outlaw unsafe initializations

Universal and Uniform Initialization or List Initialization

- Modern C++ initialization syntax uses { }-list-based notation
- Compiler won't accept initializer values that will be narrowed

```
double x{}; // ok: x initialized to 0

int a{1'000}; // ok
char b{a};    // error: int -> char will narrow

char c{1'000}; // error: narrowing for chars
char d{48};    // ok
```

Old-Style (or, C-Style) Casts


```
double x = 10.23;
int i = (double)x; // C notation
int j = double(x); // C++ function cast notation
```

static_cast Operator


```
int num{10}, den{3};

// C++ static_cast operator converts
// one type to another related type
double result = static_cast<double>(num)/den;
```

Why Use `static_cast` Operator?

- Spotting C-style casts is difficult – an ugly cast operator name will help tools find (dangerous) casts
- C-style casts allow you to cast any type to pretty much any other type

```
int const ci {10};
int *pi;
pi = (int*)(&ci); // ok but living dangerously!!!
pi = static_cast<int*>(&ci); // ERROR
```

Range-for Loop

- C++ takes advantage of notion of half-open range to provide simple loop over all elements of a sequence

```
int arr[] {5, 7, 9, 4, 6, 8};
for (int element : arr) {
  std::cout << element << ' ';
}
std::cout << "\n";

int *pi{arr};
for (int element : pi) { // ERROR
  std::cout << element << ' ';
}
```