School of Computing & Communications | Lancaster University

SCC.111 Software Development – Lecture 27: Reflections on C++

Adrian Friday, Hansi Hettiarachchi and Nigel Davies

Introduction

- Last lecture, we looked at:
  - Objects as function parameters
  - C++ references and initializer lists
  - Composition

- Today we’re going to reflect on what we've learned
  - Discuss the principles of C/C++
  - Reflect on some of its strengths and weaknesses
  - Discuss its suitability for different application domains

The Story so far…

- We have seen that
  - There is diversity in programming languages.

  - We create scalable programs through **modularity**.
  - **Object Oriented** languages help us to do this.
  - We learned about the core OO mechanisms:
    - classes, methods, attributes, object instances, encapsulation, constructors, destructors.
    - libraries, namespaces, composition and some examples of OO.

  - We discussed throughout the key principles of programmers taking **responsibility** for their code and writing clean, modular code for other programmers to use.

  - **We gained our first experience of OO programming in C++**

So…

- **What do you think of C++?**
  - What are the strengths of C++?

Join at menti.com | use code 7719 3468

Some reflections on C++

C++: an octopus made by nailing extra legs onto a dog.

[Steve Taylor, Dartmouth]

I invented the term Object-Oriented, and I can tell you I did not have C++ in mind.

[Alan Kay]

C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg.

[Bjarne Stroustrup]

I think maybe the guy who invented C++ doesn't know the difference between increment and excrement.

[anon]

C++ and scalability…

- **Is C++ scalable?**

  - OO languages are generally good at this, as we've discussed.
  - Classes promote reuse, collaboration and assist testing as your code grows.
  - Libraries and namespaces minimize complexity too.

  - C++ does not have the same level of platform independence as many other languages though.

  - C++ typically compiles to machine code…
    - **So you can't share executables or libraries across different architectures…**

C++ and performance…

- **Is C++ efficient?**
  - **YES**.
  - C++ compilers are very mature and optimize your code heavily.
  - Hardware can be directly controlled from C/C++ which enables the programmer to create hardware optimizations too.
  - C++ also uses *tree-shaking* at link time, to remove redundant code and reduce the size of the final binary program.
  - **Programs written in C++ typically use less CPU, less RAM and less storage* than other languages.**

** Although the code density of other languages can be higher, resulting in smaller programs at scale*

C++ and simplicity…

- **Is C++ simple?**
  - Many underlying principles are simple (loops, variables, conditional, classes)
  - But C++ does lack consistency, with many exceptions to the rules…
  - How do we refer to a variable?
    - Statically by name, by object reference, by a pointer?
    - Different APIs may use each of these
    - Programmer must choose which to use, and they each have their own behaviour
  - Where is our variable?
    - Where we define a variable has huge impact of its behaviour too
    - Global (BSS), Local (Stack) or dynamically allocated (Heap)
    - What happens when these go out of scope / are freed?

C++ and simplicity….

So there are nine different types of variable before we even begin!

	Local (stack)	Global (bss)	Dynamic (heap)
Direct access	?	?	?
Object reference	?	?	?
Pointer	?	?	?

C++ and simplicity…..

- **Is C++ simple?**

There are at least two ways to call a constructor, depending if you want a pointer or not!

```
Car myCar("white");
Car *mycar = new Car("white");
```

There are even multiple ways to initialise attributes in classes…

```
Class Car(string s) : colour(s) {}
Class Car(string s) {
   colour = s;
}
```

C++ and simplicity……

- **Is C++ simple?**

And this is before we get to the really complex bits!

- Inheritance, polymorphism, traits, templating...
- Each of which adds yet another dimension of complexity
- Often with feature interactions with other parts of the language.

The result is a language that places a high cognitive load on developers…

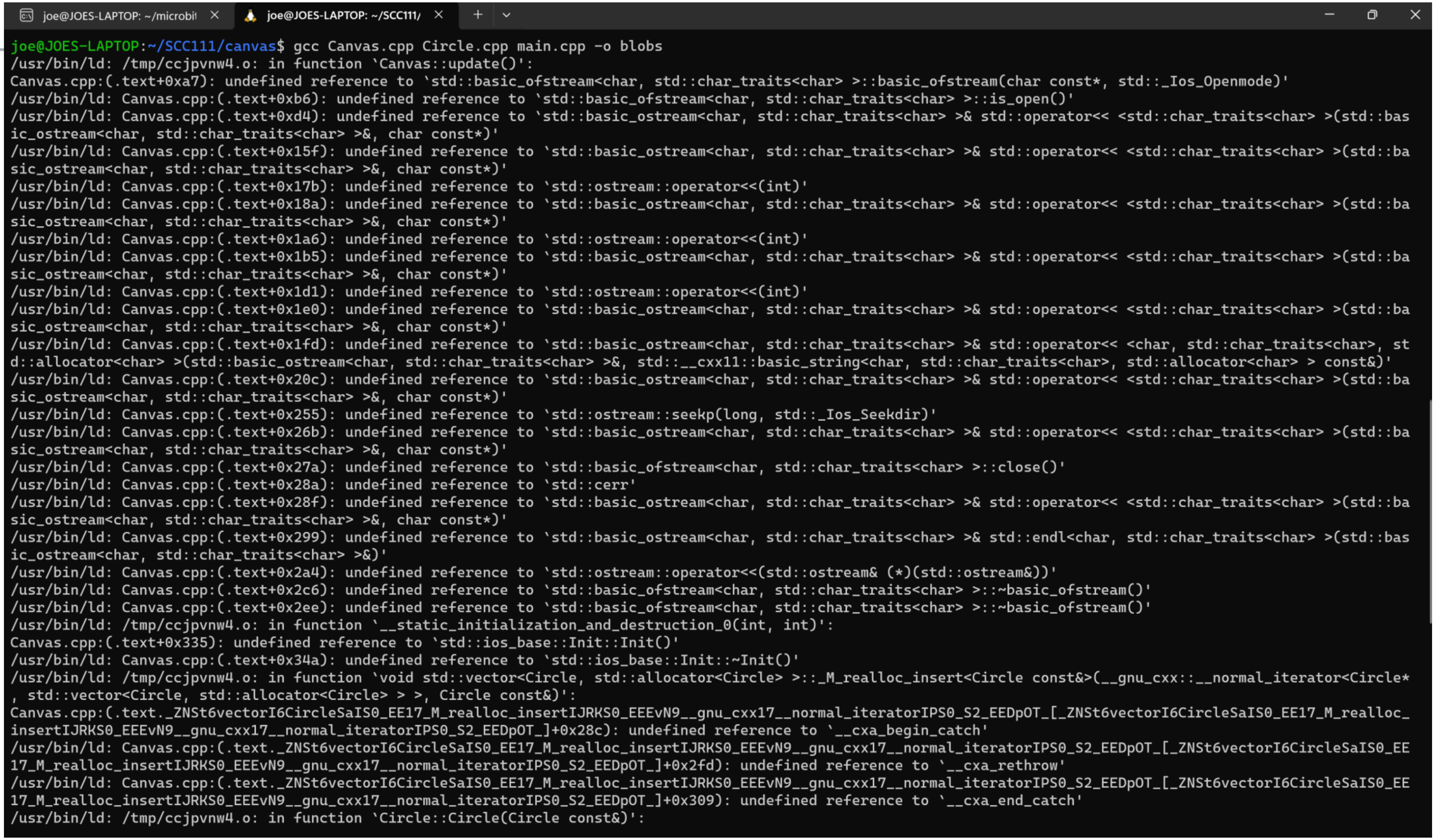
C++ and safety…

- **Is C++ safe?**

C++ and safety… Type Safety

- **Is C++ safe?**
  - C++ has quite strong type checking. The compiler will identify attempts to assign incompatible types to your variables… This make C++ largely type safe.
  - This is not true for pointers however. C/C++ will allow you to cast any pointer type to a memory location and trusts the programmer to be correct.
  - The C++ compiler will also enforce encapsulation, and restrict access to private attributes and methods. However…

C++ code checking…



```
joe@JOES-LAPTOP:~/SCC111/canvas$ gcc Canvas.cpp Circle.cpp main.cpp -o blobs
/usr/bin/ld: /tmp/ccjpvnw4.o: in function `Canvas::update()':
Canvas.cpp:(.text+0xa7): undefined reference to `std::basic_ofstream<char, std::char_traits<char> >::basic_ofstream(char const*, std::_Ios_Openmode)'
/usr/bin/ld: Canvas.cpp:(.text+0xb6): undefined reference to `std::basic_ofstream<char, std::char_traits<char> >::is_open()'
/usr/bin/ld: Canvas.cpp:(.text+0xd4): undefined reference to `std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::bas
ic_ostream<char, std::char_traits<char> >&, char const*)'
/usr/bin/ld: Canvas.cpp:(.text+0x15f): undefined reference to `std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::ba
sic_ostream<char, std::char_traits<char> >&, char const*)'
/usr/bin/ld: Canvas.cpp:(.text+0x17b): undefined reference to `std::ostream::operator<<(int)'
/usr/bin/ld: Canvas.cpp:(.text+0x18a): undefined reference to `std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::ba
sic_ostream<char, std::char_traits<char> >&, char const*)'
/usr/bin/ld: Canvas.cpp:(.text+0x1a6): undefined reference to `std::ostream::operator<<(int)'
/usr/bin/ld: Canvas.cpp:(.text+0x1b5): undefined reference to `std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::ba
sic_ostream<char, std::char_traits<char> >&, char const*)'
/usr/bin/ld: Canvas.cpp:(.text+0x1d1): undefined reference to `std::ostream::operator<<(int)'
/usr/bin/ld: Canvas.cpp:(.text+0x1e0): undefined reference to `std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::ba
sic_ostream<char, std::char_traits<char> >&, char const*)'
/usr/bin/ld: Canvas.cpp:(.text+0x1fd): undefined reference to `std::basic_ostream<char, std::char_traits<char> >& std::operator<< <char, std::char_traits<char>, st
d::allocator<char> >(std::basic_ostream<char, std::char_traits<char> >&, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)'
/usr/bin/ld: Canvas.cpp:(.text+0x20c): undefined reference to `std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::ba
sic_ostream<char, std::char_traits<char> >&, char const*)'
/usr/bin/ld: Canvas.cpp:(.text+0x255): undefined reference to `std::ostream::seekp(long, std::_Ios_Seekdir)'
/usr/bin/ld: Canvas.cpp:(.text+0x26b): undefined reference to `std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::ba
sic_ostream<char, std::char_traits<char> >&, char const*)'
/usr/bin/ld: Canvas.cpp:(.text+0x27a): undefined reference to `std::basic_ofstream<char, std::char_traits<char> >::close()'
/usr/bin/ld: Canvas.cpp:(.text+0x28a): undefined reference to `std::cerr'
/usr/bin/ld: Canvas.cpp:(.text+0x28f): undefined reference to `std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::ba
sic_ostream<char, std::char_traits<char> >&, char const*)'
/usr/bin/ld: Canvas.cpp:(.text+0x299): undefined reference to `std::basic_ostream<char, std::char_traits<char> >& std::endl<char, std::char_traits<char> >(std::bas
ic_ostream<char, std::char_traits<char> >&)'
/usr/bin/ld: Canvas.cpp:(.text+0x2a4): undefined reference to `std::ostream::operator<<(std::ostream& (*)(std::ostream&))'
/usr/bin/ld: Canvas.cpp:(.text+0x2c6): undefined reference to `std::basic_ofstream<char, std::char_traits<char> >::~basic_ofstream()'
/usr/bin/ld: Canvas.cpp:(.text+0x2ee): undefined reference to `std::basic_ofstream<char, std::char_traits<char> >::~basic_ofstream()'
/usr/bin/ld: /tmp/ccjpvnw4.o: in function `__static_initialization_and_destruction_0(int, int)':
Canvas.cpp:(.text+0x335): undefined reference to `std::ios_base::Init::Init()'
/usr/bin/ld: Canvas.cpp:(.text+0x34a): undefined reference to `std::ios_base::Init::~Init()'
/usr/bin/ld: /tmp/ccjpvnw4.o: in function `void std::vector<Circle, std::allocator<Circle> >::_M_realloc_insert<Circle const&>(__gnu_cxx::__normal_iterator<Circle*
, std::vector<Circle, std::allocator<Circle> > >, Circle const&)':
Canvas.cpp:(.text._ZNSt6vectorI6CircleSaIS0_EE17_M_realloc_insertIJRKS0_EEEvN9__gnu_cxx17__normal_iteratorIPS0_S2_EEDpOT_[_ZNSt6vectorI6CircleSaIS0_EE17_M_realloc_
insertIJRKS0_EEEvN9__gnu_cxx17__normal_iteratorIPS0_S2_EEDpOT_]+0x28c): undefined reference to `__cxa_begin_catch'
/usr/bin/ld: Canvas.cpp:(.text._ZNSt6vectorI6CircleSaIS0_EE17_M_realloc_insertIJRKS0_EEEvN9__gnu_cxx17__normal_iteratorIPS0_S2_EEDpOT_[_ZNSt6vectorI6CircleSaIS0_EE
17_M_realloc_insertIJRKS0_EEEvN9__gnu_cxx17__normal_iteratorIPS0_S2_EEDpOT_]+0x2fd): undefined reference to `__cxa_rethrow'
/usr/bin/ld: Canvas.cpp:(.text._ZNSt6vectorI6CircleSaIS0_EE17_M_realloc_insertIJRKS0_EEEvN9__gnu_cxx17__normal_iteratorIPS0_S2_EEDpOT_[_ZNSt6vectorI6CircleSaIS0_EE
17_M_realloc_insertIJRKS0_EEEvN9__gnu_cxx17__normal_iteratorIPS0_S2_EEDpOT_]+0x309): undefined reference to `__cxa_end_catch'
/usr/bin/ld: /tmp/ccjpvnw4.o: in function `Circle::Circle(Circle const&)':
```

C++ and safety… Pointers

- **Is C++ safe?**
  - C/C++ places no restrictions on the assignment and dereferencing of pointers.
  - Programmers are free to attempt to read/write any locations in memory

  - **ANY** part of a C++ program can access **ANY** memory associated with the program.
  - It is left to the operating system to protect other running programs…

  - Pointers can be used to accidentally (or intentionally) alter memory. Even variables that are declared private.

C++ and safety… Security

- **Is C++ safe?**
  - This leaves programs written in C++ open to some quite significant security risks…

  - String/array overrun attacks
  - Stack frame rewriting attacks

Let's see an example…

Summary

- Today we learned that:

  - C++ is suited to high-performance, low-level application domains, where performance is often of critical importance.

  - Hence, we often find C++ being used in domains where this is true, such as:

Operating Systems Development
Embedded Systems
Computer Games Development