Managed by Fermi Research Alliance, LLC for the U.S. Department of Energy Office of Science

Session 1
Basics of C++

Chris Jones

art/LArSoft Course

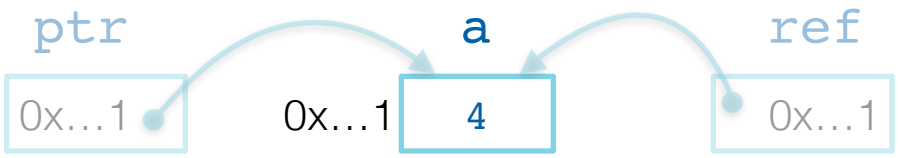
03 August 2015

What we will cover

- Covering just those topics which are known *gotchas*
- Pointers and References
- Const
- Function calls
- Compiling and Linking

Non-References/Pointers

- Non reference/pointer variables store values in memory

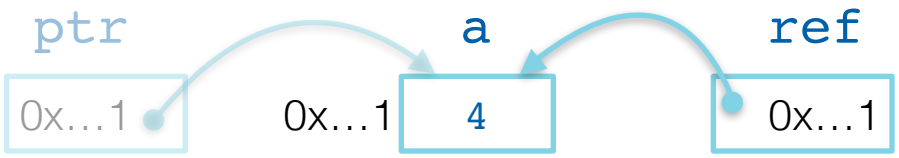



```
int a = 4;

int& ref = a;

int* ptr = &a;
```

Variable Name	Memory Address	Value
a	0x…1	4
ref	—	4
ptr	0x…2	0x…1

References

- References are aliases to existing memory
  - References do not have a separate memory address

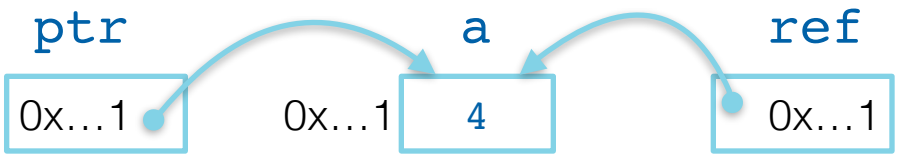



```
int a = 4;

int& ref = a;

int* ptr = &a;
```

Variable Name	Memory Address	Value
a	0x…1	4
ref	—	4
ptr	0x…2	0x…1

Pointers

- Pointers store memory addresses in memory
  - Pointers do have an assigned memory address




```
int a = 4;

int& ref = a;

int* ptr = &a;
```

Variable Name	Memory Address	Value
a	0x…1	4
ref	—	4
ptr	0x…2	0x…1

Differences Between Pointers and References

- Pointers can change which memory address they store

```
int a = 4;

int& ref = a;

int* ptr = &a;

int b = 5;

ptr = &b;
```

Differences Between Pointers and References

- Pointers can *point* to nothing
  - Think of a pointer as a container that can hold 0 or 1 item

```
int* ptr = nullptr;
```

Differences Between Pointers and References

- Pointers must be dereferenced to get the associated value

```
int a = 4;

int& ref = a;
int* ptr = &a;

int b = ref; //same as b = a;
int c = *ptr; // same as c = a;
```

Changing values

- Pointers and references can modify the associated value

```
int a = 4;

int& ref = a;
ref = 5; //a == 5

int* ptr = &a;
*ptr = 6; //a == 6
```

`const` keyword

- `const` tells the compiler the value is not allowed to change

```
int const a = 4;

a = 5; //compiler error
```

- const may appear to the left or right of the type name

```
const int a = 4;
```

`const` with Pointers and References

- Pointers and references can refer to `const` variables but must be `const` themselves

```
int const a = 4;

int const& ref = a;
int const* ptr = &a;

//the following will not compile
int& non_const_ref = a;
int* non_const_ptr = &a;
```

`const` with Pointers and References continued

- Pointers and references can refer to `non-const` variables but still be `const` themselves

```
int a = 4;

int const& ref = a;
int const* ptr = &a;

//the following will not compile
ref = 6;
*ptr = 7;

//the following will work
a = 8; //ref == 8 and *ptr == 8
```

`const` and Pointers

- `const` can be used several ways with pointers
  - **const pointer**: memory address can not be changed

```
int * const ptr = &a;
ptr = &b; //compiler error
*ptr = 6; //OK
```

  - **pointer to const**: can not change associated value

```
int const * ptr = &a;
ptr = &b; //OK
*ptr = 6; //compiler error
```

  - **const pointer to const**: can not change address or value

```
int const * const ptr = &a;
```

`const` and Pointers 2

- Easiest to remember by reading from right to left
  - **const pointer**: memory address can not be changed

```
int * const ptr
```

  - **pointer to const**: can not change associated value

```
int const * ptr
```

  - **const pointer to const**: can not change address or value

```
int const * const ptr
```

Function arguments

- Functions arguments can be passed by
  - **copying the value**
    - changing the value in the function does not affect the original variable

```
void foo( int a);
```

  - **reference**
    - changing the value does change the value of the original variable

```
void foo( int& a);
```

  - **pointer**
    - changing the value does change the value of the original variable

```
void foo( int* a);
```

Function arguments continued

- Functions arguments can be passed by
  - **const references**
    - it is not possible to change the value of the argument

```
void foo( int const& a);
```

  - **const pointer**
    - it is not possible to change the value of the argument

```
void foo( int const* a);
```

Building Code

- Code building with C++ has two phases
  - Compiling
  - Linking

- cetbuildtools handles both steps for you

Compiling

- The compiler reads the source file and generates an *object file*

- An object file contains
  - CPU instructions for the functions in the source file
  - Names of the functions and global variables from the source file

- Object files normally end with the suffix **.o** or **.os**

Linking

- The linker creates a *shared library* from a group of object files

- A shared library contains
  - All the cpu instructions from the group of object files
  - Names of all the functions and global variables from the object files

- A shared library can be *linked* to other shared libraries
  - Linking allows a function in one shared library to use functions or global variables from another shared library

- shared libraries normally end with the suffix **.so**
  - On mac OS X they can also end in **.dylib**

Compilation Errors

- Compilation errors occur when there are syntactical problems with your source code

- Remedy is to change your source code

Linking Errors

- Linking errors happen when the linker cannot find functions or global variables it needs

- Remedy
  - If the function is in your source file double check the spelling of its name
  - If the function is one you created be sure you defined it in a source file
  - If the function comes from somewhere else, be sure to link to the library containing the function

Exercises

- The info on all the exercises of the course is available at
  - https://cdcvs.fnal.gov/redmine/projects/art-larsoft-course/wiki/Instructions_for_Each_Session
- If you have trouble accessing the URL
  - create a temporary directory and change to it
    - `mkdir tmp; cd tmp`
  - checkout the code
    - `git clone http://cdcvs.fnal.gov/projects/art-workbook-alcourse alcourse`
  - change to the directory
    - `cd alcourse`
  - get the correct version of the file
    - `git checkout -b work origin/August2015`
  - follow instructions in the file **README**