CSCI 390 – Special Topics in C++

Lecture 3

8/28/18

Time To Turn Off Cell Phones

Intrinsic Types
Floating Point Types

```
// main.cpp:
#include <iostream>

#include "Helper.h"

int main(void)
{
    float f1 = 1.0f;
    double f2 = 2.0;

    auto a1 = 1.0f;
    auto a2 = 2.0;

    std::cout << DUMPVAR(f1) << std::endl;
    std::cout << DUMPVAR(f2) << std::endl;

    std::cout << DUMPVAR(a1) << std::endl;
    std::cout << DUMPVAR(a2) << std::endl;

    return 0;
}
```

Type	Constants
`float`	1.0f
`double`	1.0
long double	1.0L

```
Console:
Variable: f1, Type: float, Length: 4, Address: 0x7ffc18eeb498, Value: 1
Variable: f2, Type: double, Length: 8, Address: 0x7ffc18eeb4b0, Value: 2
Variable: a1, Type: float, Length: 4, Address: 0x7ffc18eeb49c, Value: 1
Variable: a2, Type: double, Length: 8, Address: 0x7ffc18eeb4b8, Value: 2


...Program finished with exit code 0
Press ENTER to exit console.
```

Intrinsic Types
enum Types

main.cpp:
```
#include <iostream>

#include "Helper.h"

int main(void)
{
    enum eYesNo : unsigned char
    {
      Yes = 1,
      Maybe,
      No = 0,
      Duh = 'a' + 1u,
    };

    eYesNo IsCSCI390 = Yes;
    eYesNo IsOnlineCourse = No;
    auto WillItRain = Maybe;
    auto Unsure = Duh;
    char cUnsure = Duh + 1u;

    std::cout << DUMPVAR(IsCSCI390) << std::endl;
    std::cout << DUMPVAR(IsOnlineCourse) << std::endl;
    std::cout << DUMPVAR(WillItRain) << std::endl;
    std::cout << DUMPVAR(Unsure) << std::endl;
    std::cout << DUMPVAR(cUnsure) << std::endl;

    return 0;
}
```

Console:
```
Variable: IsCSCI390, Type: main::eYesNo, Length: 1, Address: 0x7ffe53acfb69, Value: 1
Variable: IsOnlineCourse, Type: main::eYesNo, Length: 1, Address: 0x7ffe53acfb6a, Value: 0
Variable: WillItRain, Type: main::eYesNo, Length: 1, Address: 0x7ffe53acfb6b, Value: 2
Variable: Unsure, Type: main::eYesNo, Length: 1, Address: 0x7ffe53acfb6c, Value: 98
Variable: cUnsure, Type: char, Length: 1, Address: 0x7ffe53acfb6d, Value: c


...Program finished with exit code 0
Press ENTER to exit console.
```

Standard Types

//main.cpp:

```
#include <iostream>
#include <cstdint>

#include "Helper.h"

int main(void)
{
    std::cout << DUMPTYPE(int8_t) << std::endl;
    std::cout << DUMPTYPE(int16_t) << std::endl;
    std::cout << DUMPTYPE(int32_t) << std::endl;
    std::cout << DUMPTYPE(int64_t) << std::endl;

    std::cout << DUMPTYPE(uint8_t) << std::endl;
    std::cout << DUMPTYPE(uint16_t) << std::endl;
    std::cout << DUMPTYPE(uint32_t) << std::endl;
    std::cout << DUMPTYPE(uint64_t) << std::endl;

    std::cout << DUMPTYPE(uint_fast8_t) << std::endl;
    std::cout << DUMPTYPE(uint_fast16_t) << std::endl;
    std::cout << DUMPTYPE(uint_fast32_t) << std::endl;

    return 0;
}
```

See https://en.cppreference.com/w/cpp/header/cstdint

Console:

```
Type: signed char, Length: 1
Type: short, Length: 2
Type: int, Length: 4
Type: long, Length: 8
Type: unsigned char, Length: 1
Type: unsigned short, Length: 2
Type: unsigned int, Length: 4
Type: unsigned long, Length: 8
Type: unsigned char, Length: 1
Type: unsigned long, Length: 8
Type: unsigned long, Length: 8


...Program finished with exit code 0
Press ENTER to exit console.
```

C++ Versions

Year	C++ Standard	Informal Name
1998	ISO/IEC 14882:1998	C++98
2003	ISO/IEC 14882:2003	C++03
2011	ISO/IEC 14882:2011	C++11
2014	ISO/IEC 14882:2014	C++14
2017	ISO/IEC 14882:2017	C++17
2020	Not assigned.	C++20

- C++03 fixed problems identified in C++98.
- C++11 (14882:2011) included many additions to both the core language and the standard library.
- C++14 was a small extension to C++11, featuring mainly bug fixes and small improvements.
- C++17 is a major revision.

<identifier>

- An identifier is an arbitrarily long string of case sensitive letters, underscores and digits starting with at least one underscore or letter.
- Underscores often prefix system variables and should be avoided.

<lvalue> vs <rvalue>

- A <lvalue> has <type>, value and memory.
  - e.g., a <variable>
- A <rvalue> has <type> and value.
  - e.g., an <expression>
- You can store a <rvalue> into a <lvalue>.
  - Think of it this way:
    <lvalue> = <rvalue>

Variables

- Variables are a user-friendly name for memory.
- Their name is an <identifier>.
- They are a <lvalue>.
  - Variables have a type, value and memory.
  - Variables hold a value of its type in its memory.
- Variables must be declared before using them.
  - Variables should be initialized at declaration.
- Variables have a life cycle (scope).

Declarations
Simple Variable Declaration

- Simple variables syntax:
  <type> <identifier> [<rvalue initializer>];
  - <identifier> is the variable name.
  - Choose type based on use.
- Examples:

```
uint8_t Month;
uint8_t DayOfMonth = 0u;
uint16_t Year{0u};
```

Declarations

`typedef`/`decltype`

- If several variables have similar usage, consider declaring a declaring a type.
- `typedef` syntax:

```
typedef <existing type> <new type>;
```

- Type names are <identifiers>.
- Sometimes the existing type is buried deep inside include files. Use `decltype` to access underlying type of a <rvalue> (variable or expression).

```
typedef decltype(<rvalue>) <new type>;
```

Declarations

`auto` Type

- auto declaration syntax:

  `auto` <identifier> <rvalue initializer>**`;`**

- Type is inferred from <rvalue initializer>, which is required.
- decltype is similar to auto. auto is equivalent to:

  `decltype(`<rvalue initializer>**`)`** <identifier> <rvalue initializer>**`;`**

Declarations
Simple Type Declaration

//main.cpp:

```
#include <iostream>
#include <cstdint>

#include "Helper.h"

int main(void)
{
   uint8_t Month{0u};
   typedef uint8_t tMonth;
   tMonth mm{1u};
   typedef decltype(mm) tMonth2;
   tMonth2 mm2 = 0x61u;
   decltype(mm2) mm3{mm2 + 1u};

   std::cout << DUMPTYPE(tMonth) << std::endl;
   std::cout << DUMPTYPE(tMonth2) << std::endl;

   std::cout << DUMPVAR(Month) << std::endl;
   std::cout << DUMPVAR(mm) << std::endl;
   std::cout << DUMPVAR(mm2) << std::endl;
   std::cout << DUMPVAR(mm3) << std::endl;

   return 0;
}
```

Console:

```
Type: unsigned char, Length: 1
Type: unsigned char, Length: 1
Variable: Month, Type: unsigned char, Length: 1, Address: 0x7ffc70ba5217, Value:
Variable: mm, Type: unsigned char, Length: 1, Address: 0x7ffc70ba5218, Value:
Variable: mm2, Type: unsigned char, Length: 1, Address: 0x7ffc70ba5219, Value: a
Variable: mm3, Type: unsigned char, Length: 1, Address: 0x7ffc70ba521a, Value: b


...Program finished with exit code 0
Press ENTER to exit console.
```

Variable Life Cycle

- Each { … } pair creates a scope.
- A variable's life extends from its declaration to the end of it's scope.
- A global variable's life cycle is the entire program.

//main.cpp:

```
#include <iostream>
#include <cstdint>

#include "Helper.h"

uint32_t Level = 0u;

int main(void)
{
  std::cout << "1) " << DUMPVAR(Level) << std::endl;
  uint32_t Level = 1u;
  std::cout << "2) " << DUMPVAR(Level) << std::endl;
  {
    std::cout << "3) " << DUMPVAR(Level) << std::endl;
    uint32_t Level = 2u;
    std::cout << "4) " << DUMPVAR(Level) << std::endl;
    std::cout << "5) " << DUMPVAR(::Level) << std::endl;
  }
  std::cout << "6) " << DUMPVAR(Level) << std::endl;

  return 0;
}
```

Console:

```
1) Variable: Level, Type: unsigned int, Length: 4, Address: 0x602224, Value: 0
2) Variable: Level, Type: unsigned int, Length: 4, Address: 0x7ffc80a320b8, Value: 1
3) Variable: Level, Type: unsigned int, Length: 4, Address: 0x7ffc80a320b8, Value: 1
4) Variable: Level, Type: unsigned int, Length: 4, Address: 0x7ffc80a320bc, Value: 2
5) Variable: ::Level, Type: unsigned int, Length: 4, Address: 0x602224, Value: 0
6) Variable: Level, Type: unsigned int, Length: 4, Address: 0x7ffc80a320b8, Value: 1


...Program finished with exit code 0
Press ENTER to exit console.
```

`const` Qualifier

- A variable that should not change its scope should include the `const` qualifier in its declaration. This is called const correctness.
- Constant variables must be initialized.
- The C++ compiler will flag any attempt to change its value.

//main.cpp:

```
#include <iostream>
#include <cstdint>

#include "Helper.h"

int main(void)
{
  const double pi{3.141592653589793};
  std::cout << DUMPVAR(pi) << std::endl;

  pi = 3.14159265358979323846;
  std::cout << DUMPVAR(pi) << std::endl;

  return 0;
}
```

Console:

```
main.cpp: In function 'int main()':
main.cpp:12:6: error: assignment of read-only variable 'pi'
   pi = 3.14159265358979323846;
     ^
```

Reference Specifier

- An alias for a variable can be created with a reference.
  - Syntax: <type>**&** <identifier> <lvalue initializer>**;**
  - The reference specifier is **&**. It is NOT an operator.
  - The identifier is the reference name.
- References must be initialized with an <lvalue>.

//main.cpp:

```
#include <iostream>
#include <cstdint>

#include "Helper.h"

int main(void)
{
  uint32_t Var = 1u;
  std::cout << DUMPVAR(Var) << std::endl;

  uint32_t& Variable{Var};
  std::cout << DUMPVAR(Variable) << std::endl;

  Variable = 2u;

  std::cout << DUMPVAR(Var) << std::endl;
  std::cout << DUMPVAR(Variable) << std::endl;

  return 0;
}
```

Console:

```
Variable: Var, Type: unsigned int, Length: 4, Address: 0x7ffd29b8004c, Value: 1
Variable: Variable, Type: unsigned int, Length: 4, Address: 0x7ffd29b8004c, Value: 1
Variable: Var, Type: unsigned int, Length: 4, Address: 0x7ffd29b8004c, Value: 2
Variable: Variable, Type: unsigned int, Length: 4, Address: 0x7ffd29b8004c, Value: 2


...Program finished with exit code 0
Press ENTER to exit console.
```

References

- An alias for a variable can be created via a reference.
  - Syntax: <type>**&** <identifier> <lvalue initializer>**;**
  - The reference specifier is **&**. It is NOT an operator.
  - The identifier is the reference name.
- References must be initialized with an <lvalue>.

//main.cpp:

```
#include <iostream>
#include <cstdint>

#include "Helper.h"

int main(void)
{
  uint32_t& One{1u};
  std::cout << DUMPVAR(One) << std::endl;

  return 0;
}
```

Console:

```
main.cpp: In function 'int main()':
main.cpp:9:19: error: invalid initialization of non-const reference of
   type 'uint32_t& {aka unsigned int&}' from an rvalue of type 'unsigned int'
   uint32_t& One{1u};
                 ^
```

References (cont)

- An alias can be a const version of the aliased variable.
- If the aliased variable is const, then the alias must be const.

//main.cpp:

```
#include <iostream>
#include <cstdint>

#include "Helper.h"

int main(void)
{
  uint32_t Var = 1u;
  std::cout << DUMPVAR(Var) << std::endl;

  const uint32_t& Variable{Var};
  std::cout << DUMPVAR(Variable) << std::endl;

  Var = 2u;

  std::cout << DUMPVAR(Var) << std::endl;
  std::cout << DUMPVAR(Variable) << std::endl;

  return 0;
}
```

Console:

```
Variable: Var, Type: unsigned int, Length: 4, Address: 0x7ffd0b0318fc, Value: 1
Variable: Variable, Type: unsigned int, Length: 4, Address: 0x7ffd0b0318fc, Value: 1
Variable: Var, Type: unsigned int, Length: 4, Address: 0x7ffd0b0318fc, Value: 2
Variable: Variable, Type: unsigned int, Length: 4, Address: 0x7ffd0b0318fc, Value: 2

...Program finished with exit code 0
Press ENTER to exit console.
```

References (cont)

- An alias can be a const version of the aliased variable.
- If the aliased variable is const, then the alias must be const.

//main.cpp:

```
#include <iostream>
#include <cstdint>

#include "Helper.h"

int main(void)
{
  uint32_t Var = 1u;
  std::cout << DUMPVAR(Var) << std::endl;

  const uint32_t& Variable{Var};
  std::cout << DUMPVAR(Variable) << std::endl;

  Variable = 2u;

  std::cout << DUMPVAR(Var) << std::endl;
  std::cout << DUMPVAR(Variable) << std::endl;

  return 0;
}
```

Console:

```
main.cpp: In function 'int main()':
main.cpp:15:12: error: assignment of read-only reference 'Variable'
   Variable = 2u;
            ^
```

References (cont)

- An alias can be a const version of the aliased variable.
- If the aliased variable is const, then the alias must be const.

//main.cpp:

```
#include <iostream>
#include <cstdint>

#include "Helper.h"

int main(void)
{
  const uint32_t Var = 1u;
  std::cout << DUMPVAR(Var) << std::endl;

  uint32_t& Variable{Var};
  std::cout << DUMPVAR(Variable) << std::endl;

  Variable = 2u;

  std::cout << DUMPVAR(Var) << std::endl;
  std::cout << DUMPVAR(Variable) << std::endl;

  return 0;
}
```

Console:

```
main.cpp: In function 'int main()':
main.cpp:12:25: error: binding 'const uint32_t {aka const unsigned int}' to reference of
  type 'uint32_t& {aka unsigned int&}' discards qualifiers
   uint32_t& Variable{Var};
                      ^
```

References (cont)

- The `auto` type specifier will include `const` qualifier.

//main.cpp:

```
#include <iostream>
#include <cstdint>

#include "Helper.h"

int main(void)
{
  const uint32_t Var = 1u;
  std::cout << DUMPVAR(Var) << std::endl;

  auto& Variable{Var};
  std::cout << DUMPVAR(Variable) << std::endl;

  Variable = 2u;

  std::cout << DUMPVAR(Var) << std::endl;
  std::cout << DUMPVAR(Variable) << std::endl;

  return 0;
}
```

Console:

```
main.cpp: In function 'int main()':
main.cpp:15:12: error: assignment of read-only reference 'Variable'
   Variable = 2u;
            ^
```

References To Constants

- References to constants have this syntax:
  `const` <type>**`&&`** <identifier> <rvalue initializer>**`;`**
- The **`&&`** is not an operator.
- **`const`** not required, but omitting considered poor form.

//main.cpp:

```
#include <iostream>
#include <cmath>

#include "Helper.h"

int main(void)
{
  const auto&& SquareRoot2{std::sqrt(2.0)};
  std::cout << DUMPVAR(SquareRoot2) << std::endl;

  return 0;
}
```

Console:

```
Variable: SquareRoot2, Type: double, Length: 8, Address: 0x7ffd23165908, Value: 1.41421


...Program finished with exit code 0
Press ENTER to exit console.
```

Casting (Intro)

- Casting changes the type of a <rvalue>.
- Casting is not perfect, but signals to the reader the author has thought through all implications and it is OK.
- Casting is risky.
  - Avoid casting between signed and unsigned.
  - Casting truncates floating point types.
- Casting is useful for initialization of types with no constants.

Casting (cont)

- C style cast:
  ((<desired type>**) (**<rvalue>**))**
- C++ style cast
  <desired type>**(**<rvalue>**)**

Casting (cont)

- Casting changes the type of a <rvalue>.

//main.cpp:

```
#include <iostream>
#include <cstdint>

#include "Helper.h"

int main(void)
{
  std::cout << DUMPVAL(0x61) << std::endl;
  std::cout << DUMPVAL(((uint8_t) (0x61))) << std::endl;
  std::cout << DUMPVAL(uint8_t(0x61)) << std::endl;

  std::cout << DUMPVAL('a') << std::endl;
  std::cout << DUMPVAL(((uint8_t) ('a'))) << std::endl;
  std::cout << DUMPVAL(uint8_t('a')) << std::endl;

  return 0;
}
```

Console:

```
Expression: 0x61, Type: int, Length: 4, Value: 97
Expression: ((uint8_t) (0x61)), Type: unsigned char, Length: 1, Value: a
Expression: uint8_t(0x61), Type: unsigned char, Length: 1, Value: a
Expression: 'a', Type: char, Length: 1, Value: a
Expression: ((uint8_t) ('a')), Type: unsigned char, Length: 1, Value: a
Expression: uint8_t('a'), Type: unsigned char, Length: 1, Value: a


...Program finished with exit code 0
Press ENTER to exit console.
```

Casting (cont)

- Casting is risky.
  - Avoid casting between signed and unsigned.
  - Casting truncates floating point types.

//main.cpp:

```
#include <iostream>
#include <cstdint>

#include "Helper.h"

int main(void)
{
  std::cout << DUMPVAL(-1) << std::endl;
  std::cout << DUMPVAL(((uint32_t) (-1))) << std::endl;
  std::cout << DUMPVAL(uint32_t(-1)) << std::endl;

  double d{23.9};
  std::cout << DUMPVAL(d + 1.0) << std::endl;
  std::cout << DUMPVAL(((uint32_t) (d + 1.0))) << std::endl;
  std::cout << DUMPVAL(uint32_t(d + 1.0)) << std::endl;

  return 0;
}
```

Console:

```
Expression: -1, Type: int, Length: 4, Value: -1
Expression: ((uint32_t) (-1)), Type: unsigned int, Length: 4, Value: 4294967295
Expression: uint32_t(-1), Type: unsigned int, Length: 4, Value: 4294967295
Expression: d + 1.0, Type: double, Length: 8, Value: 24.9
Expression: ((uint32_t) (d + 1.0)), Type: unsigned int, Length: 4, Value: 24
Expression: uint32_t(d + 1.0), Type: unsigned int, Length: 4, Value: 24


...Program finished with exit code 0
Press ENTER to exit console.
```

Casting (cont)

- Casting is useful for initialization of types with no constants.

//main.cpp:

```
#include <iostream>
#include <cstdint>

#include "Helper.h"

int main(void)
{
  uint16_t s1{uint16_t(0)};
  auto s2{uint16_t(1)};

  std::cout << DUMPVAR(s1) << std::endl;
  std::cout << DUMPVAR(s2) << std::endl;

  return 0;
}
```

Console:

```
Variable: s1, Type: unsigned short, Length: 2, Address: 0x7ffee83b05cc, Value: 0
Variable: s2, Type: unsigned short, Length: 2, Address: 0x7ffee83b05ce, Value: 1


...Program finished with exit code 0
Press ENTER to exit console.
```