COSC 151: Intro to Programming: C++

Chapter 2
The Basics

Chapter 2: Variables and Basic Types

- **Objectives**
  - We should be able to
    - Understand some basic types
    - Define Variables
    - Understand some complex types
      - References
      - Pointers (gasp!)
    - Understand the concept of `const`-ness
    - Understand that we can define our own data structures (our own types!)

C2: Primitive Built-in Types

- **C++ defines a set of primitive built in types**
  - The most common types (used all the time, more in the text)

Type	Meaning	Minimum Size
`bool`	true/false	N/A
`char`	character	8 bits
`int`	integer	32 bits
`double`	Floating point	64 bits

C2: Machine Level Representation

- A **byte** is the smallest chunk of addressable memory
- A **bit** holds **0** or **1**.
- Most modern architectures have 8 bit bytes, meaning a single byte can contain 2^8 values (0 - 256).

C2: Singed and Unsigned Types

- **Most arithmetic types can be signed or unsigned**
  - A signed type can hold a negative value, but the range of available values is reduced
  - An unsigned type cannot hold a negative value
  - Example:

    130 in binary: `10000010`, held in 8 bits (`unsigned char`)

    -126 in binary: `10000010`, held in 8 bits (`signed char`)

C2: Type Conversions

- The **type of an object** defined the data that an object might contain, and what operations that object can perform.
- Many types support **type conversions** to related types

```
bool b = 42;     // b is true
int i = b;       // i has a value 1
i = 3.14;        // i has a value 3
                 // (note:  truncation!)
double pi = i;   // pi has value 3.0
```

C2: Signed/Unsigned Expressions

- **Singed and Unsigned types don't play well together. Don't mix signed and unsigned types in an expression**

C2: Variables

- **A variable provides us with named storage that a program can manipulate.**
- **A variable definition has the form**
  - Type specifier
  - One or more names separated by comma
  - Each name may optionally have an initializer
  - Example:

```
int x = 3, y, z = 1;  // defines x, type int, with value 3,
                      // y, type int, with undefined value,
                      // and z, type int, with value 1
Sales_item item;      // defines item, type Sales_item
```

C2: Initializers

- **An initializer can be an arbitrarily complicated expression**

```
double price = 109.99; // price initialized
// discount initialized in terms of price
double discount = price * 0.16;
// sale_price initialized as result of function call
double sale_price = apply_discount(price, discount);
```

C2: Default Initialization

- **When a variable is defined without an initializer, the variable is default initialized.**
  - The value used for default initialized values depends on the *type and location* where the variable is defined.
  - Class types (user defined types) controls how types are default initialized.

```
std::string empty;  // default initialized to the empty string
Sales_item;         // default initialized Sales_item
```

C2: Declarations and Definitions

- **Variables can be declared many times**
- **Variables must be defined only once**
  - Variables are declared with the `extern` keyword
    - It's rare to see this done anymore in modern C++
- **Differentiating between declaration and definition will become more important when we discuss Functions and User Defined Types.**

C2: Identifiers

- **Identifiers are the valid names we can give variables/functions/classes**
  - Can be composed of letters, digits, and the underscore character
  - No limit to length
  - Must begin with a character or underscore
  - Cannot conflict with any language keywords (see text, Table 2.3)
  - Case-sensitive

```
int foo, foo1, FOO, FOO1;  // 4 different (valid) identifiers
int 101dalmations, price$; // invalid identifiers
```

C2: Scope

- **A scope is a part of a program in which a name has meaning**

```
int main()  // defined outside of braces in global scope
{
   int sum = 0;  // defined inside block of main, has block scope

   // val defined in for statement, has statement scope
   for(int val = 1; val <= 10; ++val)
   {
      sum += val;
   }

   std::cout << “The sum of 1 to 10 inclusive is “ << sum << std::endl;
}
```

C2: Nested Scopes

- **Scopes nest. The inner scope (nested) contains the outer scope.**

```
int reused = 42; // reused has global scope

int main()
{
   int unique = 0; // unique has block scope
   // prints 42 0, as expected
   std::cout << reused << " " << unique << std::endl;

   int reused = 0; // new local object named reused,
                   // "hides" global resused
                   // this is almost always a bad idea!

   // prints 0 0, the local reused object is printed
   std::cout << reused << " " << unique << std::endl;

   // prints 42 0, by explicitly using the global reused
   // using the scope resolution operator (::)
   std::cout << ::reused << " " << unique << std::endl;
   return 0;
}
```

C2: Compound Types

- A **Compound Type** is a type that is defined in terms of another type.
- There are two compound types of consequence: **References**, and **Pointers**

C2: References

- **A reference defines an alternative name for an object.**
  - A reference must be bound when defined
  - Once assigned, a reference cannot be bound to another object

```
int val = 1024;
int& val_ref = val; // val_ref refers to (is another name for) val
int& bad_ref;       // error:  a reference must be initialized
```

C2: References (contd)

- **A reference is an alias**
  - Any operation on a reference are actually operations on the object to which the reference refers.

```
int val = 1024;
int& val_ref = val;
val_ref -= 24; // shorthand for val_ref = val_ref - 24;
               // val now holds the value 1000
int val2 = val_ref;  // val2 holds the value 1000
```

C2: References (contd)

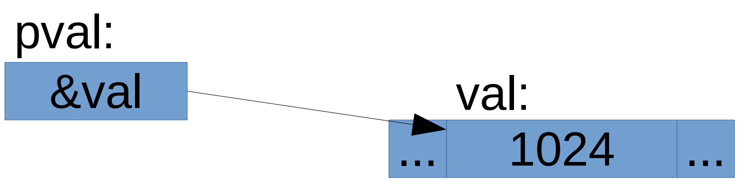
- **A reference must be bound to an appropriate type**

```
int val = 1024;
double& val_ref = val; // error:  a double& cannot bind to int
```

  - An appropriate type doesn't always mean the exact type – more on this later...

C2: Pointers

- **A pointer "points to" – holds the address of another object.**



- **Unlike a reference, a pointer is not simply an alias**

```
int val = 1024;
int* pval = &val; // use the "address of" operator (&)
                  // to get the address of val,
                  // which is held as the value in  pval
```

C2: Pointers (contd)

- **A pointer gives indirect access to an object**
  - It uses some unfamiliar syntax to do so

```
int val = 1024;
int* pval = &val;  // pval holds the address of val

// prints “1024 at address <?>”
// where <?> is the memory address of val
// use the dereference operator (*) to get the value of a pointed to object
cout << *pval << “ at address “ << pval << std::endl;

*pval -= 24; // dereference operator and subtraction
             // val now holds the value 1000
int val2 = *pval;  // val2 holds the value 1000
```

C2: Pointers (contd)

- **Pointers can be rebound**

```
int val = 1024;
int* pval = &val;  // pval holds the address of val
int foo = 500;
pval = &foo;       // pval now holds the address of foo
```

- **A value pointer can**
  - Point to an object
  - Point to a location immediately past the end of an object
  - Be a null pointer (`nullptr`) indicating it is not bound to (does not point to) any object
- **Any other pointer value is invalid.**

C2: Pointers (contd)

- **A pointer must be bound to an appropriate type**

```
int val = 1024;
double* pval = &val; // error:  a double* cannot bind to int*
```

  - An appropriate type doesn't always mean the exact type – more on this later…
- **Uninitialized pointers can be a source of obscure and hard to track down bugs.**
  - Initialize all pointers!

C2: Symbols with Multiple Meanings

- **Reference**
  - & following a type and part of a declaration

    ```
    int& val_ref = val;
    ```
- **Address-Of**
  - & used in an expression

    ```
    int* pval = &val;
    ```
- **Pointer**
  - * following a type and part of a declaration

    ```
    int* pval = &val;
    ```
- **Dereference**
  - * used in an expression

    ```
    int val2 = *pval;
    ```

C2: The `const` Qualifier

- **Things that are `const` cannot be changed**
  - Often we want constant values in our programs
  - Once assigned, a `const` value cannot be changed

```
const int buffer_size = 512;
const double pi = 3.14159;

buffer_size = 1024; // error cannot change const value
pi = 3.15;          // error cannot change const value
```

C2: Reference to `const`

- **A reference to `const` (`const` reference) cannot be used to change the the object to which the reference is bound**
  - This may seem to defeat the purpose of a reference
    - Actually, this is an incredibly powerful tool that is used extensively in C++ programs

C2: Reference to const (contd)

- **A const reference may refer to an object that is not const**

```
int val = 1024;
const int& cref_val = val;
cref_val = 1000;  // error, cannot modify const object
val = 1000;       // val can be changed,
                  // and cref_val is still an alias to val
// prints “1000 and 1000”
std::cout << val << “ and “ << cref_val << std::endl;
```

C2: `constexpr` and Type Aliases

- **A `constexpr` is an expression that can be evaluated at compile time**

```
constexpr int calc_buffer_size() { return 512 * 4; }
constexpr int buffer_size = calc_buffer_size();
// both statements are evaluated at compile time, not run time
```

  - An incredibly powerful tool, for the most advanced users

- **Type Aliases**
  - Another name for a type

```
typedef double wages; // now, wages is another name for double
using wages = double; // same as above, C++11 syntax
                      // easier to understand left to right reading
                      // prefer this to typedef
```

C2: auto specifier

- **The auto specifier can be used in place of a specific type, when the actual type is known by the compiler**
  - Get in the habit of using this wherever possible

```
std::string long_string = "It was the best of times, it was the w...";
// how long is it?
int len = long_string.length(); // may work, may not
                                // definitely NOT correct
unsigned int len = long_string.length();  // better,
                                          // but still may not work
// correct, but painfully verbose (see 2.5.3 for decltype)
decltype(long_string.length()) len = long_string.length();

auto len = long_string.length(); // correct AND terse, ahhh!!!!
```

C2: Defining our own Types

- **Defining types is a core concept of object oriented languages, including C++**
- **Defining types allows us to group together related data elements and a strategy for using them**

C2: Defining our own Types (contd)

- **The `Sales_data` structure**

```
struct Sales_data
{
   std::string bookNo;
   unsigned int units_sold = 0;
   double revenue = 0.0;
};
```

  - Here we use the `struct` keyword to define our type, we can also use class (interchangeably), the behavior of `class`/`struct` is *almost* 100% the same
  - The class body (block within `{}`) defines the **members** of the class
    - This class has only **data members**.

C2: Defining our own Types (contd)

- **Member variables are defined in the same we we define normal variables**
  - The initializers are used to give the data members an initial value

```
struct Sales_data
{
   std::string bookNo;          // std::string objects are "empty" when default initialized
   unsigned int units_sold = 0; // units_sold will be initialized to 0
   double revenue = 0.0;        // revenue will be initialized to 0.0
};
```

- **The data members of each object are independent, changing one `Sales_data` object doesn't affect any other `Sales_data` object.**

```
Sales_data book1, book2;
book1.revenue = 100.00; // use the dot operator to access members
bool2.revenue = 300.00;
// prints 100 300
std::cout << book1.revenue << " " << book2.revenue << std::endl;
```

C2: Defining our own Types - Usage

```
int main()
{
   Sales_data data1, data2; // two books

   // read in book information
   double price = 0; // the price of each book
   std::cin >> data1.bookNo >> data1.units_sold << price;
   // calculate revenue as a function of units * unit price
   data1.revenue = data1.units_sold * price;

   // read in second book (notice we don't redefine price)
   std::cin >> data2.bookNo >> data2.units_sold >> price;
   data2.revenue = data2.units_sold * price;


   if(data1.bookNo == data2.bookNo) // same book?
   {
      auto total_count = data1.units_sold + data2.units_sold;
      auto total_revenue = data1.revenue + data2.revenue;

      std::cout << data1.bookNo << " " << total_count << " " << total_revenue << " ";

      // protect against divide by 0
      if(total_count != 0)
         std::cout << total_revenue / total_count << std::endl;
      else
         std::cout << "(no sales)" << std::endl;
   }
   else
   {
      std::cerr << "Data must refer to the same ISBN" << std::endl;
      return -1;  // indicate failure (to the OS)
   }
};
```

C2: Writing our own Header Files

- **As our programs get bigger, putting everything in one file is not a sustainable practice**
- **We can break up our programs into multiple files**
  - Data Types should typically go into their own header

```
// Sales_data.h...
#pragma once  // prefer this to ifdef/define/endif
struct Sales_data
{
   std::string bookNo;
   unsigned int units_sold = 0;
   double revenue = 0.0;
};
```

- **Now we can use the Sales_data class throughout our program by including that file:**

```
#include ”Sales_data.h”  // include our Sales_data.h file (looks in the same directory!)
```

Final Thoughts

- **Built-in types** are types provided by the language
- **Variables** provide named storage for types that a program can manipulate
- Any given program has multiple **scopes**.
- **References** are aliases to an existing variable. `const` references are particularly interesting (and valuable)
- **Pointers** “point to” variables, allowing indirect access.
- Use `auto` wherever you can!
- Define our own types using `struct` or `class`.