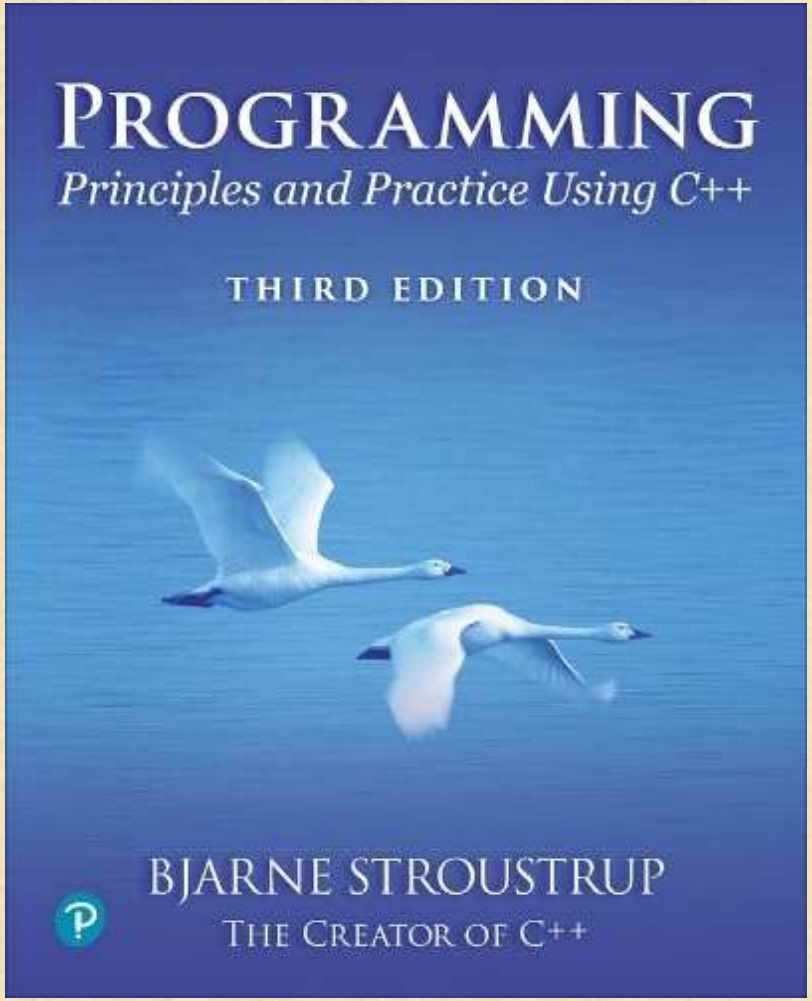
Chapter 2: Objects, Types, and Values



Fortune favors the prepared mind.
– Louis Pasteur

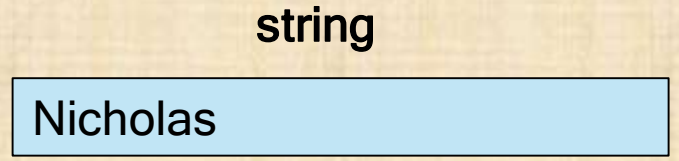
Abstract

- Most programming tasks involve manipulating data. Today, we will:
  - describe how to input and output data
  - present the notion of a variable for holding data
  - introduce the central notions of “Type” and “Type Safety”

- Strings and string I/O
- Integers and integer I/O
- Types and objects
- Type safety

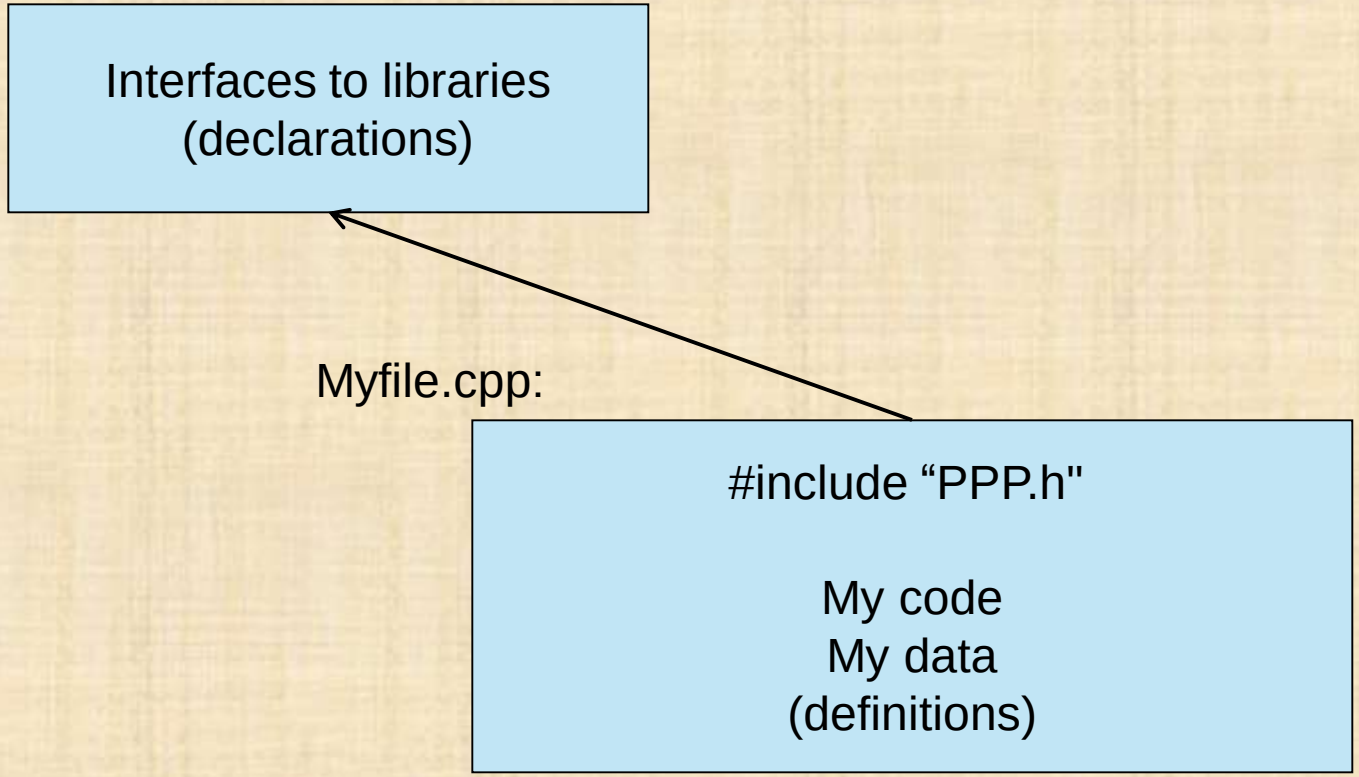
Input and output

```
// read first name:
#include "PPP.h"                    // our course header

int main()
{
  cout << "Please enter your first name (followed " << "by 'enter'):\n";
  string first_name;
  cin >> first_name;
  cout << "Hello, " << first_name << '\n';
}

// note how several values can be output by a single statement
// a statement that introduces a variable is called a declaration
// a variable holds a value of a specified type (here, we entered Nicholas)
```



Source files

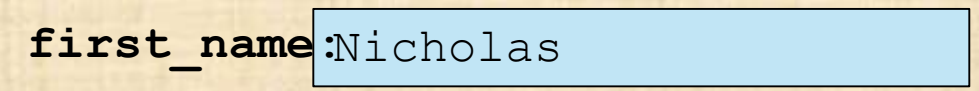
PPP.h:



- “PPP.h" is the header for our course

Input and type

- We read into a variable
  - Here, **first_name**
- A variable has a type
  - Here, **string**
- The type of a variable determines what operations we can do on it
  - Here, **cin>>first_name;** reads characters until a whitespace character is seen (“a word”)
  - White space: space, tab, newline, …
- A variable has a value
  - Here, we entered **Nicholas**

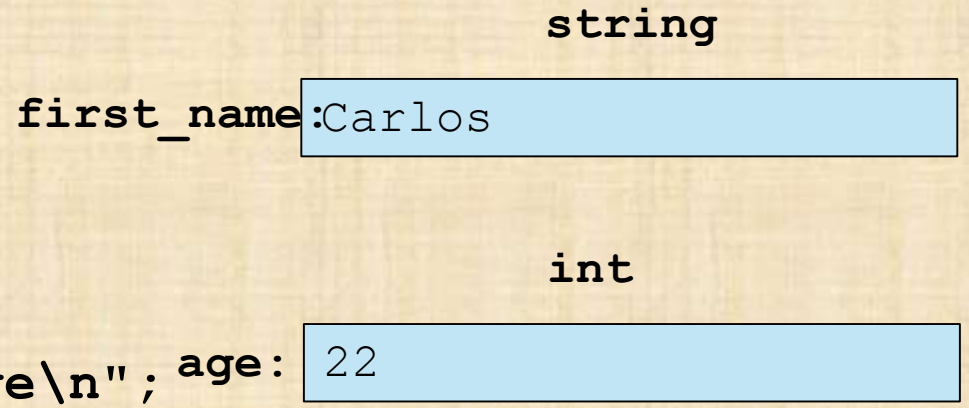


String input

```
// read first and second name:
int main()
{
 cout << "please enter your first and second names\n";
 string first;
 string second;
 cin >> first >> second;                    // read two strings
 string name = first + ' ' + second;        // concatenate strings
 separated by a space
 cout << "Hello, "<< name << '\n';
}

// I left out the #include " PPP.h" to save space and reduce
 distraction
// Don't forget it in real code
```

Integers

```
// read name and age:

int main()
{
 cout << "please enter your first name and age\n";
 string first_name;            // string variable
 int age;                      // integer variable
 cin >> first_name >> age;     // read
 cout << "Hello, " << first_name << " age " << age << '\n';
}

// I entered Carlos 22
```



Integers and Strings

- Strings
  - **cin >>** reads a word
  - **cout <<** writes
  - **+** concatenates
  - **+= s** adds the string **s** at end
  - **++** is an error
  - **-** is an error
  - …
- Integers and floating-point numbers
  - **cin >>** reads a number
  - **cout <<** writes
  - **+** adds
  - **+= n** increments by the int **n**
  - **++** increments by **1**
  - **-** subtracts
  - …

The type of a variable determines which operations are valid and what their meanings are for that type

that's called “overloading” or “operator overloading”

Names

- A name in a C++ program
  - Starts with a letter, contains letters, digits, and underscores (only)
    - Names:
      - **x, number_of_elements, Fourier_transform, z2**
    - Not names:
      - **12x, timetomarket, main line**
    - Do not start names with underscores: **_foo**
      - those are reserved for implementation and systems entities
  - Users can't define names that are taken as keywords
    - Keywords:
      - **int, if, while, double**

Names

- Choose meaningful names
  - Abbreviations and acronyms can confuse people
    - **mtbf, TLA, myw, nbv**
  - Short names can be meaningful
    - (only) when used conventionally:
      - **x** is a local variable
      - **i** is a loop index
  - Don't use overly long names
    - Ok:
      - **partial_sum**
        element_count
        staple_partition
    - Too long:
      - **the_number_of_elements**
        remaining_free_slots_in_the_symbol_table

Simple arithmetic

```
// do a bit of very simple arithmetic:

int main()
{
 cout << "please enter a floating-point number: ";    // prompt
 for a number
 double n;                                            // floating-point
 variable
 cin >> n;
 cout << "n == " << n
      << "\nn+1 == " << n+1                           // '\n'
 means "a newline"
      << "\nthree times n == " << 3*n
      << "\ntwice n == " << n+n
      << "\nn squared == " << n*n
      << "\nhalf of n == " << n/2
      << "\nsquare root of n == " << sqrt(n)     // sqrt() is a
```

A simple computation

```
int main()          // inch to cm conversion
{
 const double cm_per_inch = 2.54;   // number of centimeters per
 inch
 int length = 1;                                  // length in inches
 while (length != 0)                              // length == 0 is used
 to exit the program
 {                                  // a compound statement (a block):
 { … }
      cout << "Please enter a length in inches: ";
      cin >> length;
      cout << length << "in.  = "
              << cm_per_inch*length << "cm.\n";
 }
}
```

Types and literals

- Built-in types
  - Boolean type
    - **bool**
  - Character types
    - **char**
  - Integer types
    - **int**
      - **and short** and **long**
  - Floating-point types
    - **double**
      - and **float**
- Standard-library types
  - **string**
  - **complex<Scalar>**

- Boolean literals
  - **true false**
- Character literals
  - **'a', 'x', '4', '\n', '$'**
- Integer literals
  - **0, 1, 123, -6, 034, 0xa3**
- Floating point literals
  - **1.2, 13.345, .3, -0.54, 1.2e3, .3F**
- String literals **"asdf"**, **"Howdy, all y'all!"**
- Complex literals
  - **complex<double>(12.3,99)**
  - **complex<float>(1.3F)**

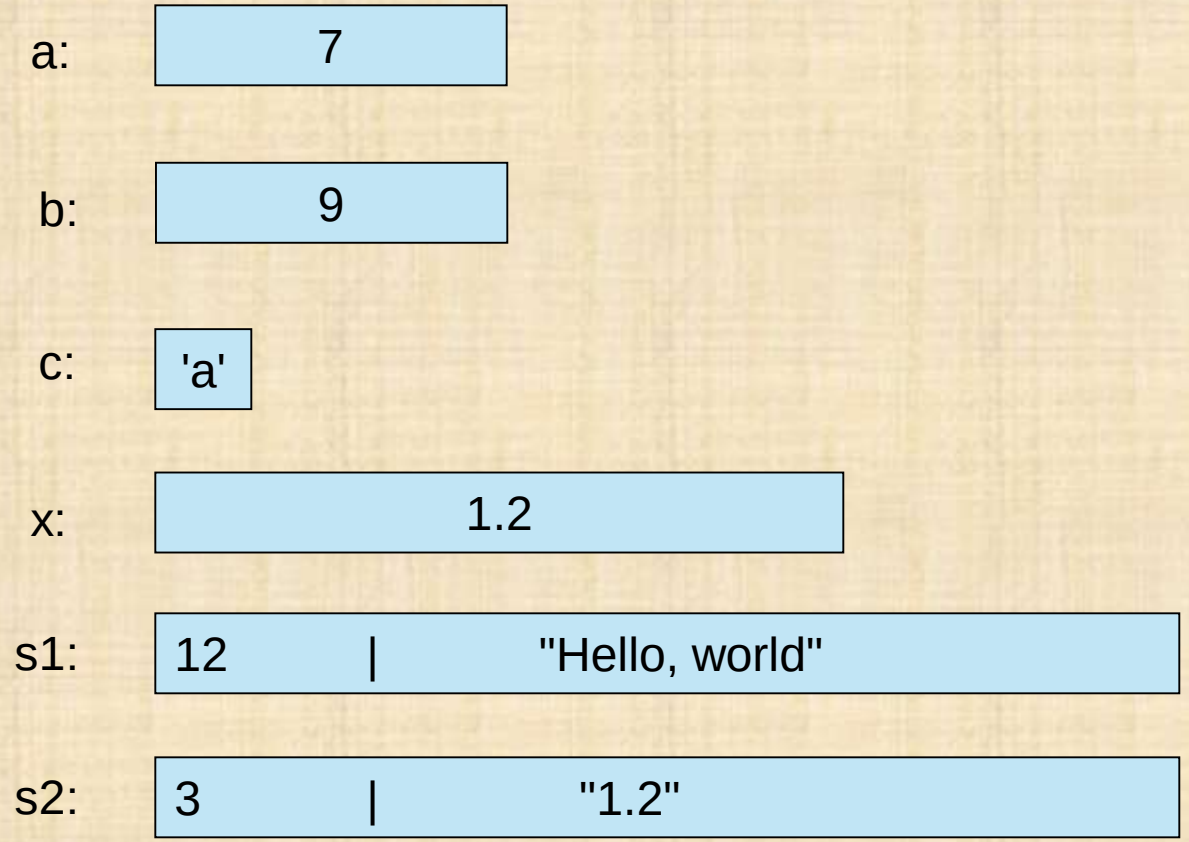
If (and only if) you need more details, see the book!

Types

- C++ provides a set of types
  - E.g. **bool**, **char**, **int**, **double**
  - Called "built-in types"
- C++ programmers can define new types
  - Called "user-defined types"
  - We'll get to that eventually
- The C++ standard library provides many more types
  - E.g. **string**, **vector**, **complex**
  - Technically, these are user-defined types
    - they are built using only facilities available to every user

Declaration and initialization

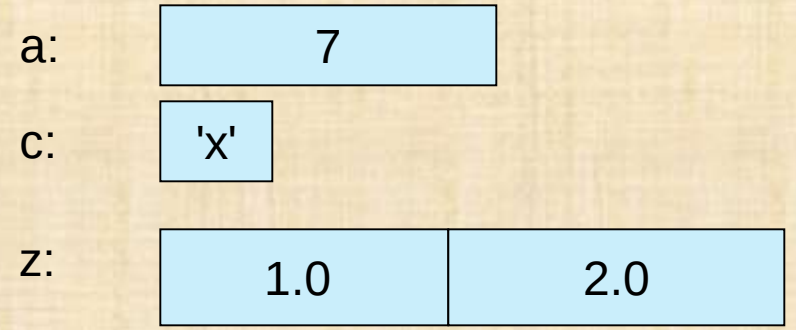
```
int a = 7;

int b = 9;

char c = 'a';

double x = 1.2;

string s1 = "Hello, world";

string s2 = "1.2";
```

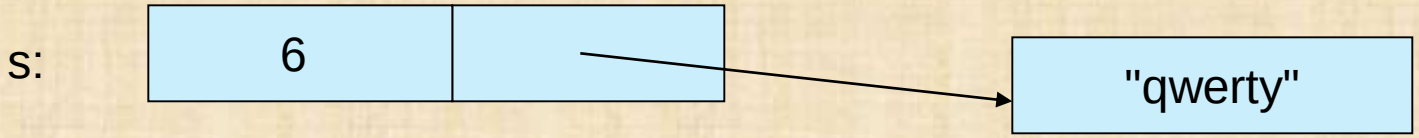


Objects

- An object is some memory that can hold a value of a given type
- A variable is a named object
- A declaration names an object

```
int a = 7;
char c = 'x';
complex<double> z {1.0,2.0};
string s = "qwerty";
```





Type safety

- Language rule: type safety
  - Every object will be used only according to its type
    - A variable will be used only after it has been initialized
    - Only operations defined for the variable's declared type will be applied
    - Every operation defined for a variable leaves the variable with a valid value
- Ideal: static type safety
  - A program that violates type safety will not compile
    - The compiler reports every violation (in an ideal system)
- Ideal: dynamic type safety
  - If you write a program that violates type safety, it will be detected at run time
    - Some code (typically "the run-time system") detects every violation not found by the compiler (in an ideal system)

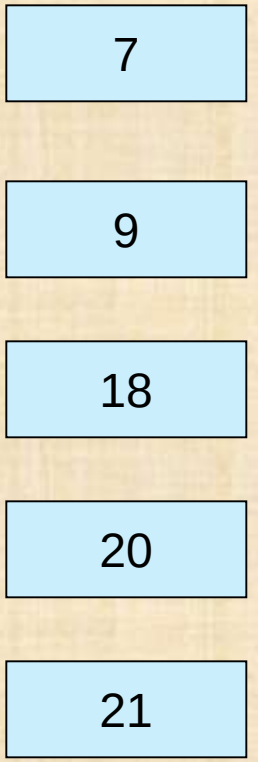
Type safety

- Type safety is a very big deal
  - Try very hard not to violate it
  - "when you program, the compiler is your best friend"
    - But it won't feel like that when it rejects code that you're sure is correct
- C++ is not (completely) statically type safe
  - No widely-used language is (completely) statically type safe
  - Being completely statically type safe may interfere with your ability to express ideas
- C++ is not (completely) dynamically type safe
  - Many languages are dynamically type safe
  - Being completely dynamically type safe may interfere with the ability to express ideas and often makes generated code bigger and/or slower
- What you'll be taught here is type safe
  - We'll specifically mention and warn against anything that is not

Assignment and increment

```
// changing the value of a variable
int a = 7;          // a variable of type int called a
                    // initialized to the integer value 7
a = 9;              // assignment: now change a's value to 9

a = a+a;            // assignment: now double a's value

a += 2;             // increment a's value by 2

++a;                // increment a's value by 1
```

a:



Type deduction: **auto**

- You can use the type of an initializer as the type of a variable
  - // **auto** means "the type of the initializer"

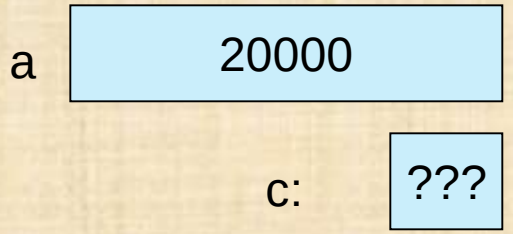
  - **auto x = 1;** // ***1*** *is an* ***int****, so* ***x*** *is an* ***int***
  - **auto y = 'c';** // ***'c'*** *is a* ***char****, so* ***y*** *is a* ***char***
  - **auto d = 1.2;** // ***1.2*** *is a* ***double****, so* ***d*** *is a* ***double***

  - **auto s = "Howdy";** // ***"Howdy"*** *is a string literal of type* ***const char[]***
    // *so don't do that until you know what it means!*

  - **auto sq = sqrt(2);** // ***sq*** *is the right type for the result of* ***sqrt(2)***
    // *and you don't have to remember what that is*

A type-safety violation ("implicit narrowing")

```
// Beware: C++ does not prevent you from trying to put a large value into a small variable
// a compiler may warn and should warn

int main()
{
 int a = 20000;
 char c = a;              // what?!!! (hope for a warning)
 int b = c;               // OK
 if (a != b)              //  != means "not equal"
     cout << "oops!: " << a << "!=" << b << '\n';     // Try it. What value does b get on your machine
 else
     cout << "Wow! We have large characters\n";
}
```



A type-safety violation (Uninitialized variables)

- Always initialize your variables (beware: "debug mode" may implicitly initialize)

```
// Beware: C++ does not prevent you from trying to use a variable before you have initialized it
// a compiler typically warns

int main()
{
 int x;                  // x gets a "random" initial value
 char c;          // c gets a "random" initial value
 double d;        // d gets a "random" initial value; not every bit pattern is a valid floating-point value
 double dd = d;         // error; some implementations can't copy invalid floating-point values
 cout << " x: " << x << " c: " << c << " d: " << d << '\n';
```

Advice

- Adhere to compiler warnings
- Always initialize your variables
- Don’t use implicit narrowing conversions

- Use **auto**
  - to avoid repeating a type name
  - to avoid implicit narrowing conversions in initializations
- Don’t use **auto** when the deduced type isn’t obvious

- Use meaningful names
  - Of appropriate length

A technical detail

- In memory, everything is just bits; type is what gives meaning to the bits
  (bits/binary) **01100001** is the int **97** is the char **'a'**
  (bits/binary) **01000001** is the int **65** is the char **'A'**
  (bits/binary) **00110000** is the int **48** is the char **'0'**

```
char c = 'a';
cout << c;      // print the value of character c, which is a
int i = c;
cout << i;      // print the integer value of the character c,
                // which is 97
```

- This is just as in "the real world":
  - What does "42" mean?
  - You don't know until you know the unit used
    - Meters? Feet? Degrees Celsius? $s? a street number? Height in inches? …

About Efficiency

- For now, don't worry about "efficiency"
  - Concentrate on correctness and simplicity of code
  - In "debug mode" we get helpful run-time checks
  - In "optimized mode" ("release", -O2) we get ***very*** fast code (compared to anything else)
- C++'s built-in types map directly to computer main memory
  - a **char** is stored in a byte
  - An **int** is stored in a word
  - A **double** fits in a floating-point register
- C++'s built-in operations map directly to machine instructions
  - An integer + is implemented by an integer add operation
  - An integer = is implemented by a simple copy operation
- C++ provides direct access to most of the facilities provided by modern hardware
- C++ help users build safer, more elegant, and efficient new types and operations using built-in types and

A bit of philosophy

- One of the ways that programming resembles other kinds of engineering is that it involves tradeoffs.
- You must have ideals, but they often conflict, so you must decide what really matters for a given program.
  - Type safety
  - Run-time performance
  - Ability to run on a given platform
  - Ability to run on multiple platforms with same results
  - Compatibility with other code and systems
  - Ease of construction
  - Ease of maintenance
- Don’t skimp on correctness or testing
- By default, aim for type safety and portability

Another simple computation

```
int main()
  // inch to cm and cm to inch conversion
{
  const double cm_per_inch = 2.54;
  int val;
  char unit;
  while (cin >> val >> unit) {          // keep reading
        if (unit == 'i')                // 'i' for inch
              cout << val << "in == " << val*cm_per_inch << "cm\n";
        else if (unit == 'c')           // 'c' for cm
              cout << val << "cm == " << val/cm_per_inch << "in\n";
        else
              return 0;                 // terminate on a "bad unit", e.g. 'q'
  }
}
```

ISO C++ standard revisions

- All language standards are updated occasionally
  - Often every 5 or 10 years
  - The ISO C++ standard is updated every 3 years
- The latest standard has the most and the nicest features
  - Currently, C++23
- The latest standard is not 100% supported by all compilers
  - GCC (Linux), Clang (Mac) and Microsoft C++ are quite good
    - See https://en.cppreference.com/w/cpp/compiler_support for endless details
    - Clang is behind on module support, hence **PPPheaders.h**
  - Other implementations (many) vary
- Essentially all correct 1990 C++ code still compile and run correctly
  - But we can write better C++ today; progress happens

The next lecture

- Will talk about expressions, statements, debugging, simple error handling, and simple rules for program construction