CSCI251/CSCI851 Autumn-2020
Advanced Programming (**S2e**)

C++ Foundations V:

Control structures, loops, and typing

Outline

- Control structures.
- Repetition structures: Loops.
- `typedef`, `using`, and `auto` type.
- Static variables in functions.

Control structures: if

```
if (Boolean expression is true)
  statement;
```

For example:

```
if ( age >= 18 )
  cout << "You must vote!" << endl;
```

If there is more than one line you use { … }, and it's often a good idea using it anyway.

Control structures: if-else

```
if (Boolean expression is true)
   statement;
else
   other statement;
```

For example:

```
if ( age >= 18 )
  cout << "You must vote" << endl;
else
  cout << "You cannot vote" << endl;
```

Compound boolean expressions

- AND: &&

```
if (age>=18 && countryCode==61)
```

- OR: ||

```
if (countryCode==61 || countryCode==64)
```

Control structures: switch

- The “if” statement is good for Boolean tests, where there are only two possible outcomes.
- For multiple outcomes, the “switch - case” structure may be more suitable.

```
switch(variable)
{
  case 1:
            actions;
            break;
  case 2:
            actions;
            break;
  .  .  .
            break;
  default:
            cout << "The case is not defined" << endl;
}
```

Switch in C++17

- Switch has additional functionality in C++17…
- See, one of my preferred sources, http://en.cppreference.com/w/cpp/language/switch
- It’s a fairly minor change that supports the inclusion of an initialisation statement.
  - Most likely to be useful for declaring a variable only to be used within the switch.

```
switch (int num = randint(2); num)
{
   case 0: std::cout << "0" ; break;
   case 1: std::cout << "1" ; break;
   case 2: std::cout << "2" ; break;
}
```

Repetition statements

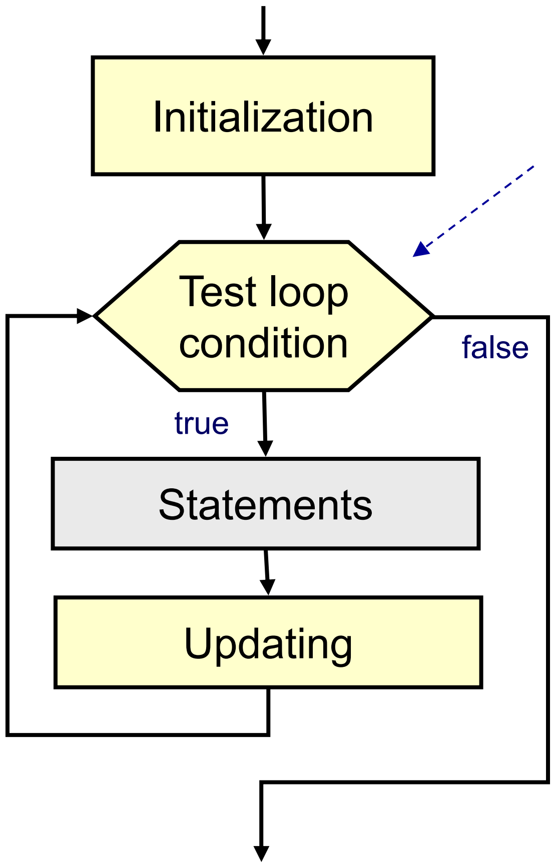
- Repetition statements are intended to implement loops that repeat an action as long as some condition remains `true` .
- There are three basic types of loop in C++:
  - – Pre-test loop `for`
  - – Pre-test loop `while`
  - – Post-test loop `do`…`while`

Repetition statements

- Although these have different syntax, all loops contain three components:
  - Initialize loop
  - Test loop condition
  - Update

Pre-test and Post-test loops

Pre-test loop



Initialization

Test loop condition

true

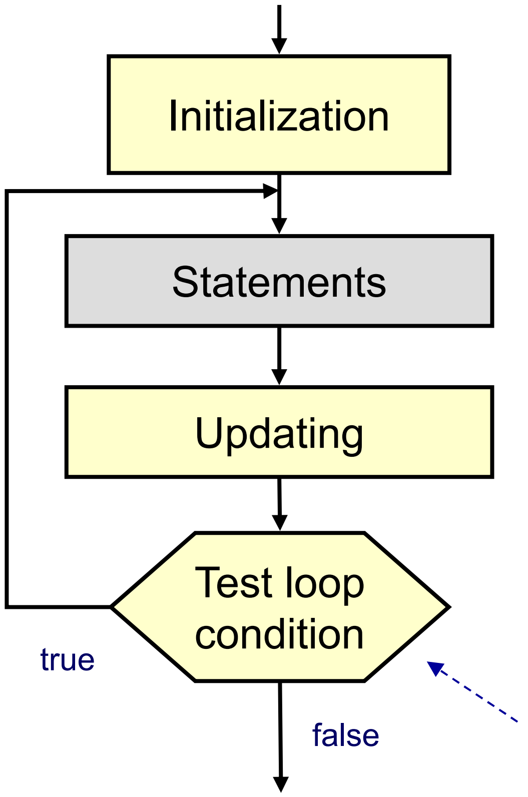
false

Statements

Updating

The loop condition is tested before the statements are executed

Post-test loop



Initialization

Statements

Updating

Test loop condition

true

false

The loop condition is tested after the statements are executed

The `while` loop

What is the limit of the sequence?

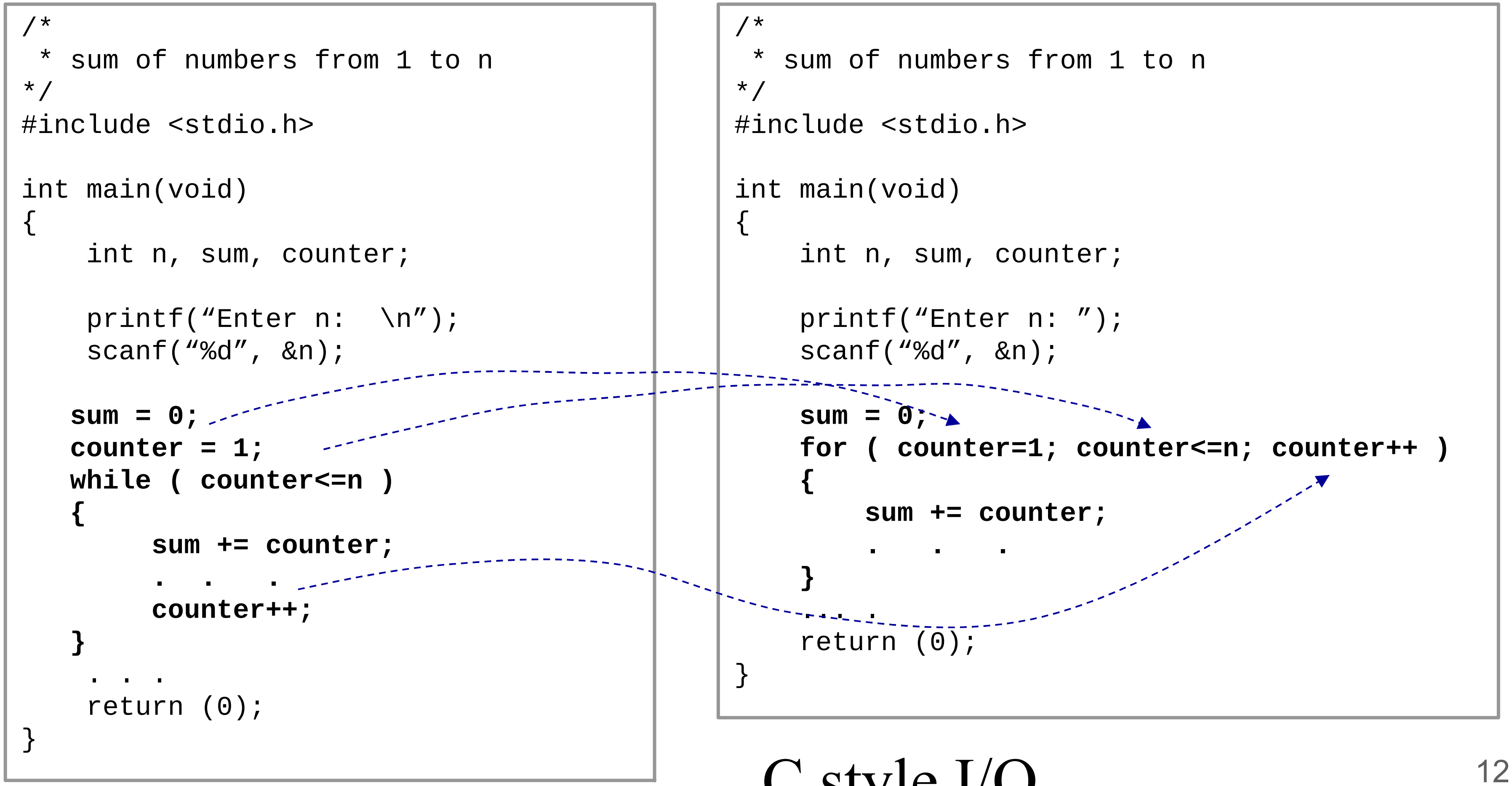
`1 + 1/2 + 1/4 + 1/8 + ...    =>     1, 1.5, 1.75, 1.875, …`

```
#define DIF  0.000001  /* Change */
.   .    .

float oldResult = 0.0 , newResult = 1.0;
float x = 2.0;

while( newResult – oldResult > DIF )
{
    oldResult = newResult;

    newResult += 1/x;
    x *= 2.0;
}

cout<< "The limit is " << newResult;
```

```
old = 0.0  new=1.0    x=2.0

new – old > 0.0001  true
old = 1.0  new=1.5    x=4.0

new – old > 0.0001  true
old = 1.5  new=1.75   x=8.0

new – old > 0.0001  true
old = 1.75  new=1.875  x=16.0
.   .   .   .

new – old > 0.0001  false
STOP

output new
```

The `for` loop

- The `for` loop is a version of a pre-test loop that has a more convenient syntax to implement a determined number of repetitions.



```
/*
 * sum of numbers from 1 to n
*/
#include <stdio.h>

int main(void)
{
    int n, sum, counter;

    printf(“Enter n:  \n”);
    scanf(“%d”, &n);

   sum = 0;
   counter = 1;
   while ( counter<=n )
   {
        sum += counter;
        .  .   .
        counter++;
   }
    . . .
    return (0);
}
```

```
/*
 * sum of numbers from 1 to n
*/
#include <stdio.h>

int main(void)
{
    int n, sum, counter;

    printf(“Enter n: ”);
    scanf(“%d”, &n);

    sum = 0;
    for ( counter=1; counter<=n; counter++ )
    {
        sum += counter;
        .   .   .
    }
    . . .
    return (0);
}
```

Variations on the `for` loop

Several initialization expressions separated by commas.

```
for( int factorial=1, counter=1; counter <= n; ++counter)
    factorial *= counter;
```

No initialization expressions.

```
for(   ; n > 0; n-- )
    printf(“*”);
```

A simple implementation of a delay (the actual delay time is platform dependent).

```
for( int counter=0; counter < 1000; counter++ ) ;
```

An infinite loop (until it is terminated inside the loop body).

```
for(  ;  ;  )
{
  . . .
}
```

The range `for` loop

- C++ supports a range for statement that allows us to step through the elements in a sequence and operate on each in the same way.

```
for( declaration : expression)
    statement
```

- The `declaration` defines the variable to be used when accessing the elements in the sequence, while `expression` is an object representing a sequence.

```
string str("This is a string");
for (char c : str)
      cout << c << endl;
```

Control structures and repetition

- We will go through some examples of control structures and repetition in one of the lecture/tutorials.
- I expect you to have a play around with them by yourself anyway.

typedef

- You can rename basic data types …

```
typedef actual_type new_name;
typedef int number;
number one, two, three;
```

- The data type `number` has the same properties as `int`.
- Using `typedef` has the potential to make code difficult to read, and its use should primarily be restricted to header files which “end users” don’t see.
  - If used sensibly it can make code tidier.

- The renaming `typedef` is probably particularly useful to get rid of deferencing operators that are likely to be around with points …
- So we could do something like …

```
typedef DataType* DataPtr;
```

- and then use the type to create pointer variables, an array for example …

```
DataPtr Index[10];
```

Alias declaration …

- There is another way of declaring an alias, one new to C++11.

```
typedef double other_double;
using OD = other_double;
OD one = 5.6;
cout << one << endl;
```

- The `using` form is likely easier to get around the right way.

`auto` typing

- This is a nice feature of the C++11 standard.
- The compiler figures out the type of something for us based on the initializer…

```
auto whatA = 5;
auto whatB = 5.6;
cout << sizeof(whatA) << sizeof(whatB);
```

- Using the `sizeof` operator lets us see the variables are at least of different sizes
- … but we can do better than that!

Checking: The `typeid` operator…

- This operator returns an object of type `type_info`, allowing us to compare types…
- To use this we need to include the header `typeinfo`, and then we can do this …

```
auto whatA = 5;
auto whatB = 5.6;
cout << typeid(whatA).name() << endl;
cout << typeid(whatB).name() << endl;
```

- Note the dot operator (`.`) is used to access a member function.
- Actually == and `!=` are likely more useful operations for this type.

Back to `auto` …

- The example we gave with an integer and a double was kind of trivial.
- The auto type specifier is most useful when the type is either hard to know or hard to write.

```
template<class T> void printall(const vector<T>& v)
{
      for (auto p = v.begin(); p!=v.end(); ++p)
            cout << *p << endl;
}
```

```
template<class T> void printall(const vector<T>& v)
{
      for (typename vector<T>::const_iterator p =
                  v.begin(); p!=v.end(); ++p)
      cout << *p << "\n";
}
```

- You can also use `auto` to grab the return type from an operation …
- So, for example and following the textbook, if we have a string and want the length we can use the member function `size()` as follows:

```
string word = "elephant";
auto length = word.size();
```

- The `size()` member function of the `string` class returns an object of `string::size_type`.

- In the range for loop described earlier it’s nice to use `auto` so the format can be the same for different sequence types.

```
string str("This is a string");
for (auto c : str)
     cout << c << endl;
```

Compounding: Referencing, `const`, and `auto`

- When we use `auto` on a reference, as in …

```
int integer = 0, &ref = integer;
auto a = ref;
```

- … the type will be that of the referenced variable, so `int` here.

- When it comes to `const` we need to consider:
  - Top-level const: Where the object is a `const`.
  - Low-level const: In compound types, with the pointer or reference being to a `const` object.

- Why make the distinction here?
  - The `const` survival differs between those two types when we, in particular here, `auto` type on a `const`, and generally when we copy…
- For example with …

```
const int ci =  integer, &cr = ci;
auto b = ci;
auto c = cr;
auto d = &integer;
auto e = &ci;
```

- … we will get b and c type `int`, d of type `int*`, and e of type `const int*`.

More `auto` (C++14)

- Since C++14 it’s possible to use `auto` rather than return on a function declaration.

```
auto function(…)
```

- The return type is now deduced from the operand of the return statement in the function.
- So

```
return int;
```

- … can be used and `int` inferred.
- It can be used to provide abstraction in Lambda expressions in C++14, and as a template parameter in C++17.

Still more on `auto`

- See …

http://en.cppreference.com/w/cpp/language/auto

- In particular, note that this is one part of C++ that has changed across C++14 and C++17.

- That website has a good history section …

http://en.cppreference.com/w/cpp/language/history

`auto` doesn’t use …

- … language recognition,
- … pattern recognition,
- … or mindreading or magic.
- Statements like …

```
auto number;
```

- … or

```
auto x;
```

- … isn’t going to work.

decltype

- Another C++11 type specifier, one with some similarities to auto, is decltype.
- This lets us base the type of a variable of a function calls return type, but initialise the variable to something else.

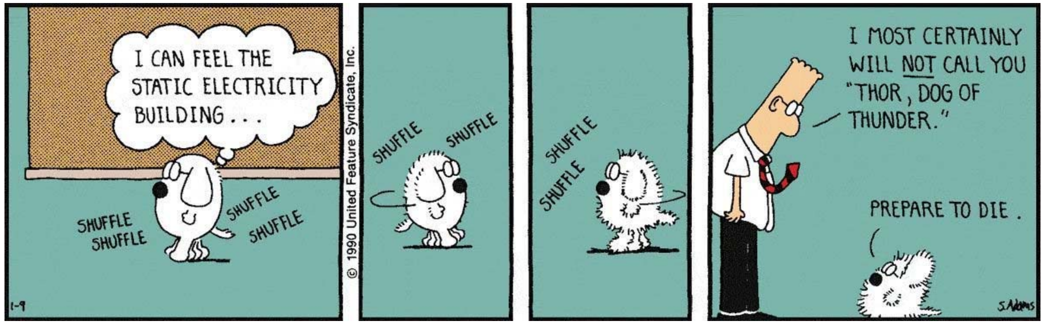
```
decltype(f()) variable = x;
```

- Here variable will have the type returned by f(), although the compiler does not itself call f().

Another qualifier: `static` …

- Be careful with this one, it has a different meaning in the context of classes.
- Static variables, declared as such in functions, persist beyond scope ({}), and aren’t re-initialised with each call to the function.

```
size_t count_calls()
{
     static size_t ctr = 0;
     return ++ctr;
}

int main()
{
     for (size_t i = 0; i < 10; ++i)
          cout << count_calls() << endl;
     return 0;
}
```


I CAN FEEL THE STATIC ELECTRICITY BUILDING . . .
SHUFFLE SHUFFLE
SHUFFLE SHUFFLE
SHUFFLE SHUFFLE
SHUFFLE SHUFFLE
I MOST CERTAINLY WILL NOT CALL YOU "THOR, DOG OF THUNDER."
PREPARE TO DIE.
© 1990 United Feature Syndicate, Inc.
1-9
S.Adams