# Week2 : Variables and Basic Types

## Scientific Computing

Blair Jamieson

University of Winnipeg

Class 2

# Outline

Objects, Types and Values

Computation

References and Pointers

# Outline

Objects, Types and Values
- Input
- Types and Objects
- Type conversions and safety
- Variables and initialization

Computation

References and Pointers

# Introduction to keyboard intput

- To read something in from the keyboard we need somwhere in the computer's memory to place it
- We call such a place an *object*
- An *object* is a region of memory with a *type* that specifies the type of information
- A named *object* is called a *variable*
- For example:
  - character strings are put into `string` variables
  - integers are put into `int` variables
  - real numbers are put into `float` or `double` variables

# Introduction to objects

- Think of object as a box in which you put a value of the object's type:
- int:

- Above represents object of type `int`, named `age`

# Example input into a string

## input_name.cpp

```
1 #include "std_lib_facilities.h"
2 int main(){
3   cout << "Please enter your name,"
4           "followed by enter:" << endl;
5   string first_name;
6   cin >> first_name;
7   cout << "Hello "<< first_name << "!" <<endl;
8   return 0;
9 }
```

- #include a bunch of std library things like string, cin, cout
- We used `cout` to print to the screen (character output)
- We used `cin` to read from the keyboard into a string (character input)

# Input and type – exercise

Type in the following program:

name_and_age.cpp

```
1 #include "std_lib_facilities.h"
2 int main(){
3   cout << "Enter your name and age:"<<endl;
4   string first_name;
5   int age;
6   cin >> first_name >> age;
7   cout << "Hello "<<first_name
8        <<" (age " << age << endl;
9   return 0;
10 }
```

# Input and type – exercise

- If you run the program and enter "Kitty 6", the output will be:
- Hello kitty (age 6)
- Note that a space end reading into string (by convention)
- Both the » and « operators are sensitive to type – can read in and output string, int, double, and other built in types
- Exercise:
  1. Add a second string `last_name` and another » operator to read it in
  2. Ask for an age in years as a `double` and print it out in months by converting multiplying by 12

# Operations and operators

Blank entries in the table, mean there is no operator for that type.

| | bool | char | int | double | string |
|---|---|---|---|---|---|
| asignment | = | = | = | = | = |
| addition | | | + | + | |
| concatenation | | | | | + |
| subtraction | | | - | - | |
| multiplication | | | * | * | |
| division | | | / | / | |
| remainder(mod) | | | % | | |
| increment by 1 | | | ++ | ++ | |
| decrement by 1 | | | -- | -- | |
| increment by n | | | += n | += n | |

# Operations and operators

Blank entries in the table, mean there is no operator for that type.

| | `bool` | `char` | `int` | `double` | `string` |
|---|---|---|---|---|---|
| add to end | | | | | += |
| decrement by n | | | -= n | -= n | |
| multiply and assign | | | *= | *= n | |
| divide and assign | | | /= | /= | |
| remainder and assign | | | %= | %= | |
| read from s into x | s>>x | s>>x | s>>x | s>>x | s>>x |
| write x to s | s<<x | s<<x | s<<x | s<<x | s<<x |
| equals | == | == | == | == | == |
| not equals | != | != | != | != | != |
| greater than | > | > | > | > | > |
| greater than or equal | >= | >= | >= | >= | >= |
| less than | < | < | < | < | < |
| less than or equal | <= | <= | <= | <= | <= |

# Some notes on operations

- Other operations are done with functions (eg. sqrt(x) for some double x)
- For integers, division truncates result, rather than rounded

```
3/2 = 1    5/2 = 2
3%2 = 1    5%2 = 1
```

- Note for integers:

```
a / b * b + a%b == a
```

# Example using operators on strings

```
int main(){
  cout << "Please enter two names" << endl;
  string first, second;
  cin >> first >> second;
  if ( first == second ) {
    cout << "That's the same name twice" << endl;
  }
  if ( first < second ){
    cout << first << " is alphabetically before " <<
        second << endl;
  }
  if ( first > second ){
    cout << first << " is alphabetically after " <<
        second << endl;
  }
  return 0;
}
```

# Assignment and initialization

An example with integers:

```
int a = 3; // a starts out with value of 3
a = 4;     // changes value of a to be 4
int b = a; // b starts out with value of 4
b = a + 5; // changes value of a to be 4+5=9
a = a + 7; // sets a to be 4+7 = 11
```

The last example above can be thought of as three steps:

1. First get value of a → 4
2. Next add 7 to 4 → 11
3. Finally put 11 into a

# Example with strings

Draw out boxes with variable contents after each line of code:

```
string a = "alpha";
a = "beta";
string b=a;
b = a + "gamma";
a = a + "delta";
```

# Example code – detect repeated words

```
1   int main(){
2     string previous = " ";
3     string current;
4     while (  cin >> current ){
5       if ( previous == current ) {
6         cout << "repeated word: " << current << endl;
7       }
8       previous = current;
9     }
10    return 0;
11  }
```

Walk through the code one line at a time and show what is in the variables previous and current.

# Example code – Output word number on repeated words

```
1    int main(){
2      int number_of_words;
3      string previous = " ";
4      string current;
5      while (  cin >> current ){
6        ++number_of_words;
7        if ( previous == current ) {
8          cout << "word number " << number_of_words
9               <<" repeated: " << current << endl;
10       }
11       previous = current;
12     }
13     return 0;
14   }
```

- each time a new word is entered, increment counter
- very similar to previous example – often useful to modify an existing code

# Variable names

- Names of variables must start with a letter, and contain only letters, numbers and underscores
- Names are case sensitive ( x and X are different names )
- It is a bad idea to write code with variables that only differ in case
- Names can't be the same as any of the kewords (see next slide)
- Pick names that are meaningful, but not too long
- Avoid names that are easy to mistype or misread (0,o,O) and (1,l,I) are sets of characters that look similar
- One strategy: use lower-case for your variables, and start with an uppercase for types you define

# C++ Keywords

```
alignas      continue      friend      register           true
alignof      decltype      goto        reinterpret_cast   try
asm          default       if          return             typedef
auto         delete        inline      short              typeid
bool         do            int         signed             typename
break        double        long        sizeof             union
case         dynamic_cast  mutable     static             unsigned
catch        else          namespace   static_assert      using
char         enum          new         static_cast        virtual
char16_t     explicit      noexcept    struct             void
char32_t     export        nullptr     switch             volatile
class        extern        operator    template           wchar_t
const        false         private     this               while
constexpr    float         protected   thread_local
const_cast   for           public      throw
```

# C++ Operator names

Note that none of these operator names can be used when declaring or defining a variable.

```
and      compl    or_eq
and_eq   not      xor
bitand   not_eq   xor_eq
bitor    or
```

# Types and objects

Draw boxes for the variable contents:

```
1     int a = 7;
2     int b = 9;
3     char c = 'a';
4     double x = 1.2;
5     string s1 = "Hello World";
6     string s2 = "1.2";
```

- String needs to store length and characters
- Amount of memory used to store values depends on type
- Meaning of particular location in memory is completely determined by its type
- For example the int value 120 would be an 'x' if it were a char

# Variables and Types

- Variables let us give names to the objects we use
- Variable types determine the meaning of the data and operations we can perform on them

| Type | Meaning | Min Size |
|---|---|---|
| bool | Boolean | NA |
| char | Character | 8 bits |
| short | short integer | 16 bits |
| int | integer | 16 bits |
| long | long iteger | 32 bits |
| long long | long integer | 64 bits |
| float | single precision real | 6 sig figs |
| double | double precision real | 10 sig figs |
| long double | extended prec. real | 10 sig figs |

# Variables and Types

- `bool` represents truth values `true` or `false`
- `char` represents a single character – same size a a single computer byte
  - there are other character types for extended character sets `wchar_t`, `char16_t`, and `char32_t`.
- `short`, `int`, and `long` all represent integers to various maximum values
- `float`, `double`, and `long double` all represent real numbers to varying precision

# Machine representation of built in types

- Computers store data as sequence of bits, each holding 0 or 1, such as: 01001111010100011110000111 0
- Computer memory dealt with in chunks of bits of sizes that are powers of 2
- Typically in "byte" sized chunks
- Most machines use 8 bit bytes, and 32 or 64 bit words
- Most computers associate a number (called an address) with each byte in memory

# A word in memory

- For 8-bit bytes and a 32-bit word, a word in memory might look like:

| address | contents of memory | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 814321 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 814322 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 814323 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 814324 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |

- address can be used to refer to any sized collection of bits
- byte at 814321, or word at 814321
- need to know type at address 814321 to give bits meaning
- as an int, above is 37387, as a float it is $\sim 5.239 \times 10^{-41}$

# Floating point types

- Typically a `float` is 32-bits, and a `double` is 64-bits and `long double` is either 96- or 128-bits.
- `float` can yield about 7 sig. digits
- `double` can yield about 16 sig. digits

# Unsigned types

- `unsigned short`, `unsigned int`, `unsigned long`, `unsigned char` can all be defined
- For example `char` has 8 bits ($2^8$ combinations):
  - a `signed char` holds values from -128 to 127
  - an `unsigned char` holds values from 0 to 255
- For example `short` with 16-bits ($2^{16}$ combinations):
  - a `short` holds values from -32768 to 32767
  - an `unsigned char` holds values from 0 to 65535

# Advice on deciding type to use

Both C and C++ define so many arithmetic types to cater to efficient use on different hardware. A few rules of thumb to decide type use:

- Use `unsigned` type when you know the value cannot be negative
- Use `int` for arithmetic, or `long` if values involved will exceed two billion
- Avoid use of `char` for arithmetic expressions
- Use `double` for floating-point computations; `float` usually does not have enough precision
- Precision of `long double` is usually unnecessary and entails considerable run-time cost

# Type conversions

Type conversions happen implicitly, but we need to understand what the effect of the conversion is.

## Arithmetic type conversions

```
1     bool b = 10;              // b is true
2     int i = b;                // i has a value 1
3     i = 3.14;                 // i has a value 3
4     int j = 3.999;            // j has a value 3
5     double pi = i;            // pi has a value 3.0
6     unsigned char c = -1; // for 8-bit char, c has
          value 255
7     signed char c2 = 256; // for 8-bit char, value of
          c2 depends on system
```

# Expressions involving unsigned types

## Example with signed and unsigned types

```
unsigned u = 10;
int i = -42;
std::cout << i + i << std::endl; // prints -84
std::cout << u + i << std::endl; // if 32-bit int
    prints 4294967295
unsigned u1 = 42, u2 = 10;
std::cout << u1 - u2 <<std::endl; // result is 32
std::cout << u2 - u1 <<std::endl; // result is
    4294967264
```

- Assigning negative number to an `unsigned` makes the value “wrap around”.
- Assigning an unsigned number to a signed one works fine.
- Be careful when doing arithmetic with unsigned types.

# Literals

The number 42 is a `literal`
Ways of representing literals:

- `bool`: `true`, `false`
- `char`: 'a', "Hello World"
- `int` types:
  - Decimal: 70, 20, 5
  - Octal: 024, 0712, 08
  - Hexidecimal: 0x14, 0x5
- `long`: 69l, 19128874L
- `unsigned`: 54u
- `float`: 3.14159, 3.14159e0, 0., 0e0, .001, 8.85e-12f

# Escape Sequences

Escape sequences begin with a \, and are used to represent characters that have special meaning in C++ :

| | | | | | |
|---|---|---|---|---|---|
| newline | \n | horizontal tab | \t | alert(bell) | \a |
| vertical tab | \v | backspace | \b | double quote | \“ |
| backslash | \\ | question mark | \? | single quote | \’ |

We can use escape sequences as if they are single characters:

```
std::cout << '\n';  // prints a newline
std::cout << "\tHello World!\n"; // prints tab follwed
    by "Hello World" and a newline.
```

# Variables

A `variable` is a a named storage location that we can manipulate with our programs. The `variable`'s type determines its size and layout in memory.

- Variable definition consists of **type specifier** followed by list of one or more named variables separated by commas, ending with a semi-colon (;)
- Variable definition may include initial value for one or more variables in the list

```
int sum =0, value, units = 0; // sum and units
    initialized to zero
Circle c; // num has type Circle (some user defined
    class)
//below, book is initialized from a string literal
std::string book("Scientific Computing");
```

# What is an Object?

C++ objects are a region of memory that contain data that has a type. An object's type can be:

- One of the built in types, eg. `int`
- A standard library class type, eg. `std::string`
- A user defined class, eg. `Chess_piece`

Note that `std::string` is a type that represents a variable-length sequence of characters. Like `std::cout`, `std::string` is defined in the `std::` namespace.

# Initialization vs. Assignment

Initialization is not assignment. Initialization happens when a variable is given a value when it is created. Assignment obliterates an object's current value and replaces that value with a new one.

## Example initializations

```
// below pi is defined and initialized before used
// to initialize twopi
double pi = 3.1415926535, twopi = pi * 2.0;
// below we call a function that calculates the sine
// of an angle in radians and pi_over_4 is initialized
// to that value.
double pi_over_4 = sin_angle_inradians( pi/4.0 );
```

# List initialization

Several different ways exist to initialize values.

## Initialization of `int` example

```
int count = 0;
int count = {0};
int count{0};
int count(0);
long double pi = 3.1415926536;
int a{pi}, b = {pi}; // results in compilation error!
int c(pi), c = pi; // compiles, but value is truncated
```

Use of { and } in initialization is called **list initialization**, and is new in C++ 11. It has the property of not allowing initialization if it leads to loss of information.

# A note on unititialized variables

- Uninitialized variables can cause run-time problems.
- Therefore, it is recommended that every object of built-in type be initialized.

# Static Typing

- C++ is a `statically typed` language, meaning it checks types at compile time. The type of an object constrains the operations that the object can perform. The C++ compiler checks whether the operations we write are supported by the types we use.
- Static checking means that the type of every object we use must be known to the compiler.
- Therefore, we must declare the type of a variable before we can use it.
- Variables must be defined exactly once, but may be declared many times.

```cpp
extern int i;  // declares but does not define i
int j;         // declares and defines j
extern double pi=3.14;  // initializing pi overides extern, defining pi
```

This will become important if we want to use a variable defined in one file of code and used in another.

# Scope of a name

- At any particular place in a program, each name in use refers to a specific entity: a variable, function, type. The name is allowed to be reused to refer to different objects at different points in the program.
- A reused name will by default use the one in the local scope, rather than the one from an outer block of code.

# Scope example 1

## Scope in a simple program

```
1 #include <iostream>
2 int main(){
3   int sum = 0;
4   for (int val = 1 ; val <= 10 ; ++val )
5      sum += val;
6   std::cout<<"Sum of 1 to 10 is "<<sum<<std::endl;
7   return 0;
8 }
```

Three names were defined: `main`, `sum`, and `val`. `main` is in the **global scope**, and accessible from anywhere in the program. `sum` and `val` are in the **local scope** of the block of code of the main function.

# Scope example 2

Bad style to use local variable of same name as global

```
#include <iostream>
int reused = 42; // reused has global scope
int main(){
  int unique = 0; // unique: local scope of main block
  // first use of reused prints 42 0
  std::cout << reused << " " << unique << std::endl;
  // new, local object named reused hides global:
  int reused = 0;
  // next use of reused (prints 0 0)
  std::cout << reused << " " << unique << std::endl;
  // get global version of reused prints 42 0
  std::cout << ::reused << " " << unique << std::endl;
  return 0;
}
```

Note it is a bad idea to define a local variable with the same name as a global variable that the function uses or might use.

# Outline

Objects, Types and Values

Computation

- Selecting among alternatives
- Iteration
- Functions

References and Pointers

# Objectives

- Computation is the act of producing some outputs based on some inputs
- Objectives are to express our computations:
  - Correctly
  - Simply
  - Efficiently
- Good program structure during development can minimize mistakes that will need to be corrected
- Main tools in good code structure:
  - *Abstraction*: Hide details that we don't need behind convenient interfaces. eg. std library algorithm sort( b, e ), where b and e are beginning and end of containers to sort
  - *Divide and conquer*: Take large problems and divide them into several smaller problems

# Expressions

- An expression computes a value from a number of operands
- Expressions can be: a literal value (10, '1', 3.14, "Hello", ... ) or a variable name
- example (draw the variable contents in boxes):

```
int length = 20;
int width = 40;
int area = length * width;
```

- `length` is the name of the object of type int that holds the value 20
- Sometimes length refers to the box that holds the value (*lvalue*)
- Sometimes length refers to the value in the box (*rvalue*)
- Use the normal mathematical precedence rules to make more complicated formulas:

```
int perimeter = 2*(length + width);
```

# Defining constants

- Variables that are initialized to a value that we do not want to be changed put the `const` qualifier before their `type`.
- `const` variables might be the size of a buffer or a physical constant.
- Because value of `const` object can't be changed, it must be initialized.

```
1 const int minEntries;  //error: must give const a value
2 const int maxEntries = 1000;
3 nmaxEntries = 500;  //error: cant write to const object
4 int iMax = maxEntries;  //ok: copies 1000 into iMax
```

- By default `const` objects are local to file of code.
- To allow them to be used in other files use:
  `extern const int maxEntries = 1000;`
- To use `extern const` defined in another file, declare it:
  `extern const int maxEntries;`

# Operators

| Operator | Name | Comment |
|---|---|---|
| f(a) | function call | pass a to f as argument |
| ++lval | pre-increment | increment and use incr. |
| --lval | pre-decrement | decrement and use decr. |
| !a | not | result is bool |
| -a | unary minus | makes value - of original |
| a*b | multiply | |
| a / b | divide | |
| a % b | modulo (remainder) | for int only |
| a+b | add | |
| a-b | subtract | |
| out << b | write b to out | out is an `ostream` |
| in >> b | read from in to b | in is an `istream` |

# Operators continued

| Operator | Name | Comment |
|---|---|---|
| a<b | less than | result is bool |
| a<=b | less than or equal | result is bool |
| a>b | greater than | result is bool |
| a>=b | greater than or equal | result is bool |
| a==b | equal | result is bool |
| a && b | logical and | result is bool |
| a \|\| b | logical or | result is bool |
| lval = a | assignment | don't confuse with == |
| lval *= a | coumpound assignment | lval = lval * a |

- we used lval as short for "value that can appear on left-hand sige of assignment"
- There are also compound assignments for /=, %=, +=, and -=

# Operator notes

- Note that $a < b < c$ means $(a < b) < c$, and $a < b$ evaluates to `true` or `false`
- Then it checks if `true`<c or `false`<c
- Therefore $a < b < c$ **does not** check if b is between a and c
- unstead can use $(a < b)$ && $(b < c)$ to test that

# Example code – warning about int arithmetic

Code to convert Celsius to Fahrenheit ( $f = 9/5c + 32$ ):

```
1    double dc;
2    cout << "Enter a temperature in Celsius:"<<endl;
3    cin >> dc;
4    double df = 9/5 * dc + 32; // error!
5    cout <<"Temperature in Fahrenheit is "<<df<<endl;
```

- Unfortunately $9/5 = 1$ in integer arithmentic
- instead change the calculation line to:

```
double df = 9.0/5 * dc +32;
```

# Statements – and the empty statement

- A statement is a set of expressions terminated by a semi-colon
- Analogy to english: "Man eating tiger", vs "Man-eating tiger"
- Semicolon lets compiler know what we mean. A line with just a semi-colon is an empty statement (means do nothing).
- Consider this code:

```
1         if ( x==5 );
2         { y = 3; }
```

- Above code, the first ; after the if statement means that "nothing" is executed if x is equal to five.
- Setting y = 3 will happen regardless of the value of x
- Most likely meant to write:

```
1         if ( x == 5 ) {
2           y = 3 ;
3         }
```

# The `if` statement

Used to pick among alternative codes to execute:

```
int main(){
  int a=0;
  int b=0;
  cout<<"Please enter two integers"<<endl;
  cin >> a >> b;
  if ( a < b ) { // condition
    // first alternative (if a<b)
    cout << "max( "<<a<<", "<<b<<") is "<<b <<endl;
  } else {
    // second alternative (if a>=b)
    cout << "max( "<<a<<", "<<b<<") is "<<a <<endl;
  }
  return 0;
}
```

# Another code using `if`

Consider – is anything wrong with this code?

```
// convert inches to cm or vice-versa
int main(){
  const double cm_per_inch = 2.54;
  int length = 1;
  char unit =0;
  cout <<"Enter a length followed by unit"
         " (c or i):"<<endl;
  cin >> length >> unit;
  if ( unit == 'i' ){
    cout<<length<<" in == " << length*cm_per_inch
        <<" cm"<<endl;
  } else {
    cout<<length<<" cm == " << length/cm_per_inch
        <<" in"<<endl;
  }
  return 0;
}
```

- Basically works (try it out) – but what if entered 20f, thinking that the program knew about feet?
- Best to add a check for unknown entry

# Code with additional check

```
1  // convert inches to cm or vice-versa
2  int main(){
3    const double cm_per_inch = 2.54;
4    int length = 1;
5    char unit =0;
6    cout <<"Enter length and unit (c or i):"<<endl;
7    cin >> length >> unit;
8    if ( unit == 'i' ){
9      cout<<length<<" in == " << length*cm_per_inch
10          <<" cm"<<endl;
11   } else if (unit == 'c'){
12     cout<<length<<" cm == " << length/cm_per_inch
13          <<" in"<<endl;
14   } else {
15     cout<<"Unknown unit "<<unit<<endl;
16   }
17   return 0;
18 }
```

- Also demonstrates how to choose among several alternatives
- Can string together as many `else if` as needed

# The `switch` statement

Use to select among values of a character, int or enum:

```
int main(){
  const double cm_per_inch = 2.54;
  int length = 1;
  char unit =0;
  cout <<"Enter a length and unit (c or i):"<<endl;
  cin >> length >> unit;
  switch (unit) {
    case 'i':
      cout<<length<<" in == " << length*cm_per_inch
           <<" cm"<<endl;
      break;
    case 'c':
      cout<<length<<" cm == " << length/cm_per_inch
           <<" in"<<endl;
       break;
    default:
       cout<<"Unknown unit "<<unit<<endl;
  }// end switch
  return 0;
}
```

# Notes on the `switch` statement

- The value in parentheses (unit) is compared to a set of constants (the cases)
- Each case is terminated with a `break;` statement **otheriwse it continues executing into the next case**
- If none of the cases is matched, it runs the code under `default:`
- The value in the switch must be an int, char or enum
- values in case labels must be `const`
- Values in case labels must be unique
- You can use several case labels for a single case

# The `while` loop

- Used to loop over code many times to help do sums, etc.
- eg. First computer program of David Wheeler (May 6, 1949):
- Code printed a list of squares from 0 to 99:

```
  0   0
  1   1
  2   4
  3   9
...   ...
 98 9604
 99 9801
```

- Of course C++ didn't exist back then, but lets write a code in C++ that does this

# The `while` loop

```
1  // print a table of squares 0 - 99
2  int main() {
3    int i = 0;
4    while ( i<100 ){
5      cout << i << '\t' << square(i) << endl;
6      i++;
7    }
8    return 0;
9  }
```

- square(i) function we will define later
- the 'slash-t' means put in a tab

# The `while` loop

- Notation for a while loop:

  `while ( condition ) statement;`
- So long as the `condition` evaluates to be `true`, the `statement` is executed again
- The single `statement;` can be replaced with several statements in a { block } of code betwen curly-braces

# The `for` loop

Rewrite our program to calculate squares using for loop:

```
int main(){
  for ( int i=0; i<100; ++i ){
    cout << i << '\t' << square(i) <<endl;
  }
  return 0;
}
```

The for loop above is equivalent to:

```
int i=0;  // initializer part of for loop
while ( i<100 ){ // condition part of for-loop
  cout << i << '\t' << square(i);
  ++i; // increment part of for-loop
}
```

# Notes on the `for` loop

- Use of for loop is easier to maintain and understand than a while loop – prefer to use it when you can
- Never modify the loop variable inside the body of a for loop:

```
for ( int i=0; i<100; ++i ){
  cout<<i<<'\t'<<square(i)<<endl;
  ++i; // error! (though it will compile
      fine)
}
```

- above code would increment i twice in each loop

# Functions introduction

- square(i) in previous examples is a function call
- A function is a named sequence of statements
- Here is a possible definition of square(i) function:

```
1 int square ( int x ){
2   return x * x;
3 }
4 // We can use the function by calling it:
5 int main(){
6   cout<<"square(2)="<<square(2)<<endl;
7   cout<<"square(10)="<<square(10)<<endl;
8 }
```

# Function Basics

- A *function* is a block of code with a name
- A function may take zero or more arguments (values we pass to the funciton).
- A function may return a result
- More than one function with the same name can exist, called an overloaded function, so long as they are defined with different arguments.

# Writing a function

A *function* definition consists of a *return type*, a name, a list of zero or more *parameters*, and a body.

## Function to calculate a factorial

```
1  // myfactorial( n )
2  // Function to calculate the factorial of an integer
3  // Passed argument is number to find factorial of (int
       an)
4  // Return value is a double
5  double myfactorial( int an ){
6    double retval = 1.0;
7    for (int i = 2; i <= an; i++) {
8      retval *= i;
9    }
10   return retval;
11 }
```

# Calling a function

CallFactorial.cpp

```
...
int main(){
  // print the factorial of every fifth integer from 5
      to 50
  for (int i=5; i<=50; i+=5 ){
    cout << "The factorial of "<< i
         << " is " << myfactorial( i )
         << endl;
  }
  return 0;
}
```

# What happens when you call a function?

- Function's parameters from corresponding arguments are initialized
- Execution of the *calling* function is suspended, and execution of the *called* function begins
- Function execution ends when a `return` statement is encountered, and control is transferred back to the calling function.
- A return value is transferred back to the calling function.

# Parameters and arguments

- Arguments are initializers for a function's parameters.
- The first argument initializes the first parameter, and so on
- The type and number of parameters passed to a function must match the function definition

## Example: single parameter factorial function

```
1 double ans=0.;
2 ans = factorial("fourty"); // error: wrong arg type
3 ans = factorial(); // error: too few arguments
4 ans = factorial(4,5,6); // error: too many arguments
5 ans = factorial(6.78); //ok: but 6.78 gets truncated
     to 6
```

# A function's parameter list

- Function needs to have parameter list, but it can be empty
- `void` keyword can be used in empty parameter list to be compatable with C
- Parameter list is a comma-separated list of parameters each of which looks like a declaration with a single named type
- Each parameter in list needs its own name

## Example: function parameter lists

```
int f1(){ /* ... */ } // implicit void par. list
int f2( void ){ /* ... */ } // explicit void par. list
int f3( int v1, v2 ){ /* ... */ } // error missing
    type of v2
int f4( int v1, int v2 ){ /* ... */ } // ok
```

# More notes on functions

- Function return types:
  - Most types can be used as a function return value
  - Return type can be declared `void` if no return value is needed
- Scope and lifetime of variables:
  - **Scope** of a name is the part of the program in which a name is visible
  - **Livetime** of an object is the time during the program's execution that the object exists
  - variables defined in a function are local to the block of code; they are in the local scope of the function, and their lifetime ends when the function ends.

# Why use functions?

- Makes the computations logically separate
- Makes program text clearer by naming the computations
- Makes it possible to re-use the function in more that one place
- Easier to test the code (one function at a time)

# Function declarations

- Everything we need to call a function is in the first line of its definition
- We can make a function `declaration` in a header file, that doesn't include the body of the function
- By having the `declaration` separate we can #include it in code that uses it, and look at its definition without reading all the code
- Eg. function declarationf for square(i):

```
1 int square( int x );
```

# Preprocessor directives

- The code preprocessor gets called by the `g++` compiler as a first step. The preprocessor facility looks for lines starting with # .
- #`include`, for which it replaces the line with the #`include` with the contents of the specified header file.
- #`define`, for which it defines a name as a preprocessor variable
- #`ifndef`, and #`endif`, for which it checks if the name after #`ifndef` has been defined:
  - If name is defined, proceed to the #`endif`
  - If name is not defined, proceed to the next linee after the #`ifndef`

# Header files and preprocessor directives

- When we write our own functions, and data structures, we should define them in a header file (typically with a .h suffix)
- When we write our own header files, we should use preprocessor directives to make sure it is only included in one file.

## Example header file for our factorial: myfactorial.h

```
1 #ifndef MY_FACTORIAL_H
2 #define MY_FACTORIAL_H
3 // function to calculate the factorial of an int value
4 // return result as a double
5 double myfactorial( int an );
6 #endif
```

The first time myfactorial.h is included the **#ifndef** test succeeds, and on subsequent includes it fails. We put the code for myfactorial in myfactorial.cpp. Sometimes we might group a collection of functions related to some data structures in a single set of .h, and .cpp files.

# Example using `cmath`

Several mathematical operations and constants are defined in the `cmath` header file.

## Using cmath, `CMathTest.cpp`

```cpp
#include <iostream>
#include <cmath>
const double PI = atan(1) * 4.0;
using namespace std;
int main(){
  // print out the sine of theta for each degree
  double dtheta = PI / 180.0;
  for (int i=0; i<360; i++){
    double di = i;
    double theta = dtheta * di ;
    cout<<"theta="<< di
        << " sin(theta)=" << sin( theta )
        << std::endl;
  }
  return 0;
}
```

# Functions in `cmath`

Below is a partial list of the functions in `cmath`.

| Trigonometric Functions | Hyperbolic Functions | Exp. and Abs. Functions | Power and Error Functions |
|---|---|---|---|
| `cos` | `cosh` | `exp` | `pow` |
| `sin` | `sinh` | `log` | `sqrt` |
| `tan` | `tanh` | `log10` | `cbrt` |
| `acos` | `acosh` | `exp2` | `hypot` |
| `asin` | `asinh` | `fabs` | `erf` |
| `atan` | `atanh` | `abs` | `erfc` |
| `atan2` | | | `tgamma` |
| Macro constants | | | |
| `INFINITY` | `NAN` | `HUGE_VAL` | |
| `HUGE_VALF` | `HUGE_VALL` | | |
| Rounding and remainder | | | |
| `ceil` | `floor` | `fmod` | `trunc` |
| `round` | `lround` | `llround` | `rint` |

# Another `cmath` example

Lets calculate the relativistic Lorentz factor

$$\gamma = \frac{1}{\sqrt{1 - \beta^2}} \qquad (1)$$

where $\beta = v/c$, for an object with various velocities $v$, and $c = 3 \times 10^8$ m/s is the speed of light.
The Lorentz factor determines the amount of time dilation $t' = \gamma t$, and length contraction $l' = l/\gamma$

# Another cmath example, LorentzFactor.cpp

```
...
double gamma( double beta ){
  if ( beta > 1.0 ) {
    cerr << "gamma: Error beta > 1" << endl;
    return INFINITY;
  }
  return ( 1.0 / sqrt( 1.0 - beta*beta ) );
}
int main(){
  const double c = 2999792458.0; // m/s
  const int ndivs = 20;
  double beta = 0., v = 0., gam = 0.;
  for (int i=0; i<ndivs; i++){
    beta = ( double(i) + 0.5 )/double(ndivs);
    v = beta * c;
    gam = gamma( beta );
    cout << "v=" << v << " beta=" << beta
         << " gamma=" << gamma <<endl;
  }
  return 0;
}
}
```

# Outline

Objects, Types and Values

Computation

References and Pointers

Defining your own data structures

# References

This is a preview of some material we will look at again in later chapters.
A **reference** defines an alternative name for an object. We define a reference by putting an ampersand & in front of its name.

```
int aval = 25;
int &bval = aval;   // bval refers to aval
int &cval;          // error, reference needs to be
    initialized
bval = 3;           // changes the value of aval to 3
```

- Instead of copying aval into bval, bval refers to aval.
- Once a reference is set there is no way to change what it refers to.
- A reference is an alias to an existing object.
- All operations on a reference act on the object the reference refers to.

# Reference definitions

```
int  i1 = 85,  i2 = 57;  // i1 and i2 are integers
int &r1 = i1,  r2 = i2;  // r1 refers to i1, r2 is int
int  i3 = 12, &r3 = i3;  // i3 is int, r3 refers to i3
int &r4 = i3, &r5 = i2;  // both r4, r5 are references
int &r6 = 10;  // error: initializer must be an object
double d1 = 3.14;
int &r7 = d1;  // error: initializer must be an int
    type object
```

Note that the type of the reference and the object it refers to mush be the same type.

# Pointers

- A **pointer** type holds the address to a location in memory.
- A pointer is declared by putting a * before its name
- To assign a value to a pointer, we need to get the address of an object in memory
- To get the address of an object we use the & operator

```
int aval = 99;
int *pval = &aval;  // pval holds the address of aval
// pval is a pointer to aval
```

# Possible pointer values

A pointer can:

- point to an object
- point to the location immediately past the end of an object
- it can be a null pointer, indicating it does not point to an object (yet)
- it can be invlaid; values other than the previous three are invalid

Trying to access the value of an invalid pointer can have undefined results, and possibly lead to your program crashing.

# Using pointers to access an object

When a pointer points to an object, we can use the dereference operator * to get the value of the object at the address the pointer holds.

PointerTest.cpp

```
#include <iostream>
using namespace std;
int main(){
  int aval = 99;
  int *pval = &aval;
  // pval is the address (location in memory) of an int
  // *pval yields the object pointed to by pval (99)
  cout << "The integer at address: " << pval
       << " is " << *pval << endl;
  return 0;
}
```

# Symbols with multiple meanings

Some symbols have multiple meanings that need to be determined by their context.

- & could either be used to declare a reference, get the address of a value, or as a bitwise and operator.
- * could either be used to declare a pointer, get the object at an address from a pointer, or as the multiplication operator.

```
int i = 42;
int &r = i;          // & declares reference r
int *p;              // * declares pointer p
p = &i;              // & gets address of integer i
// below & declares reference r2, * dereferences p
int &r2 = *p;
// below *p = 42 is multiplied by r2 = 42, ij = 42*42
int ij = *p * r2;
```

# Null pointers

A **null pointer** is a pointer that does not point to any object. Code can check if a pointer is null before attempting to use it. If a pointer is declared before it is initialized, it is a good idea to instead initialize it to a null pointer, in one of the following three ways:

```
// nullptr is literal meaning undefined pointer (C++11)
int *p1 = nullptr;
// directly initialize p2 from literal constant 0
int *p2 = 0;
// (must #include <cstdlib>, equivalent to *p3=0
int *p3 = NULL;
```

Modern C++ programs should prefer to use `nullptr`

# Checking a pointer is valid

So long as a pointer is valid we can use it. It is good practice to check if a pointer is not **nullptr** before using it. The following are ways of doing that:

PointerCheck.cpp

```
1  #include <iostream>
2  using namespace std;
3  int main(){
4    int aval = 99;
5    int *ptr1 = nullptr;
6    int *ptr2 = &aval;
7    // if ptr is nullptr it is 0 (false), otherwise it
         is true
8    if ( ptr1 )
9      cout << "Pointer 1 ok, aval=" << *ptr1 << endl;
10   // use not equal operator != to check if ptr2 is
         valid
11   if ( ptr2 != nullptr )
12     cout << "Pointer 2 ok, aval=" << *ptr2 << endl;
13   return 0;
14 }
```

# void Pointers

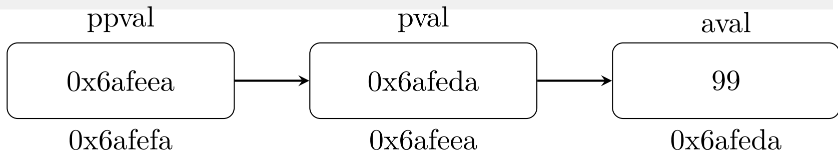
- A pointer to a place in memory, without reference to the type of object at that memory, is a **void ***.
- There are limited things that can be done with **void ***, and it is typically used to deal with memory as memory, rather than as objects at a location.

```
double pi = 3.14, *ppi = &pi;
void * vpi = &pi;
vpi = ppi;      // Ok: vpi can point to any object type
int three = 3, *pthree = &three;
vpi = pthree;  // Now vpi points to three
```

# Pointers to pointers

- A pointer is an object in memory (the address of a location in memory)
- We can store the address of the location in memory holding a pointer to another location in memory (a pointer to a pointer).

```
int aval = 99;
int *pval = &aval;  //pval points to an int
int **ppval = &pval; // ppval points to a pointer to an int
```



# References to pointers

Since a pointer is an object in memory (an address to another object), we can have a reference to it.

```
int aval = 99;
int *pval;       // pval is a pointer to int
// r is an (int*) reference to pointer pval
int * &r = pval;
// r refers to a pointer, assign &aval to r
// then pval points to aval
r = &aval;
// dereferencing r yields aval, changing aval to 0
*r = 0;
```

# Week 2 Lab assignment!

- Homework 1 given out last week due Sept. 15 by 17:00
- Homework 2 given out today is due Sept. 22 by 17:00
- Lets work on these together now!
- If you want to do these at home, are still having trouble getting tools set up to do the homework see me now!

# Pointer and reference exercises

1. Explain the main differences between pointers and references.
2. What does the following program do?

```
int i = 99;
int *pi = &i
*pi = *pi * *pi;
```

3. Explain the following definitions, and determine if they are legal. Assume `int i = 0;` has already been defined.
   3.1 `double *dp = &i;`
   3.2 `int *ip = i;`
   3.3 `int *p = &i;`

# const and Pointers and References Exercises

1. Which of the following lines of code are legal? Explain why.

```
1 int i=-1, &r=0;
2 const int i2 = i, &r2 = i;
3 const int *p1 = &i2;
4 int * const p2 = &i2;
5 const int i3 = -1, &r3=0;
6 const int * const p3 = &i2;
```

2. Explain the following def's, and identify any that are illegal.

```
1 int i, *const cp;
2 int *p1, *const p2;
3 const int ic, &r = ic;
4 const int *const p3;
5 const int *p;
```

3. Using def's in the previous prob., which below are legal?

```
i  =  ic;
p1 =  p3;
p1 = &ic;
p3 = &ic;
p2 =  p1;
ic = *p3;
```

# References to `const` objects

We can bind a reference to objects that are `const`, however the reference must also be `const`. A `const` reference cannot be used to change the object referred to (read only).

```
const int maxEntries = 1000;
//below is ok: both ref and object are const
const int &rmax = maxEntries;
rmax = 500; // error: rmax is a reference to const
//error below: non const reference to const object
int &ref2 = maxEntries;
const int &ref3 = 42;  //ok: ref3 refers to constant
const int &ref4 = maxEntries * 2; //ok
//error below: ref5 is non-const reference
int &ref5 = maxEntries * 2;
```

Note that the `const` qualifier to a reference means that the reference is to a value that is a `const`. In some sense all references are `const` as they cannot be altered once bound.

# Pointers and `const`

- A **pointer to a const** cannot be used to change the value of the object it points to
- We can only store the address to a `const` object in a pointer to `const`
- **const*** used to make a pointer that can't be reassigned

## Examples

```
const double pi = 3.141926;
double * ppi = &pi;  //error: ppi is plain pointer
const double * cppi = &pi; //ok: cppi points to const
*cppi = 42; //error: cant change const value
double aval = 99;
cppi = &aval; //ok: can use const *, point to non-const
const double *const cpcpi = &pi;
cpcpi = &aval; //err: cant change what *const points to
```

# Defining data structures

A data structure is a way to group together related data elements. Lets define a structure to describe the location and size of a rectangle. The `struct` keyword is used to define structures.

## Example, defining a `struct`

```
1 struct Box_st {
2   double xmin=0.;
3   double ymin=0.;
4   double xmax=1.;
5   double ymax=1.;
6 };
```

One thing to remember is to put a semi-colon ; at the end of `struct` declarations.

# Using data in structures

## Example, making two boxes and an enclosing box

```
1  Box_st b1, b2;
2  b1.xmin=1.5;  b1.ymin=1.5;  b1.xmax=5.5; b1.ymax=4.5;
3  b2.xmin=-1.5; b2.ymin=-1.5; b2.xmax=2.0; b2.ymax=2.5;
4  Box_st boxframe;
5  if ( b1.xmin < b2.xmin ) boxframe.xmin = b1.xmin;
6  else boxframe.xmin = b2.xmin;
7  if ( b1.ymin < b2.ymin ) boxframe.ymin = b1.ymin;
8  else boxframe.ymin = b2.ymin;
9  if ( b1.xmax > b2.xmax ) boxframe.xmax = b1.xmax;
10 else boxframe.xmax = b2.xmax;
11 if ( b1.ymax > b2.ymax ) boxframe.ymax = b1.ymax;
12 else boxframe.ymax = b2.ymax;
```