TDDE18 & 726G77

Struct and Functions

Lessons

- First lesson on Friday 14th September
- One lesson hall will be held in English (R34), all other will be held in Swedish
- The content in the lesson will be directly applicable for lab2

Labs update

- Soft deadline for lab 1 pushed until
  - Monday for group A
  - Tuesday for group B
- On lab1, there will be complementary work on:
  - Code style
  - Code duplication
  - Etc.
  - Assessment protocol on the course web site

Code style

```
int main(){cout<<”hello world”<<1+1<<endl;int
a{2};a++;cin>>a;cout<<a;}
```

This is a very bad code style. Its hard to read and understand what the program do. Code formatting and code style is very important in this course.

It is your job to write code that is easy to understand.

It is easier to teach you good code style after you have submitted a little code.

That is why we will give you extensive feedback on your own submission for lab1.

Compiling multiple files

Single file:

```
g++ filename1.cpp
```

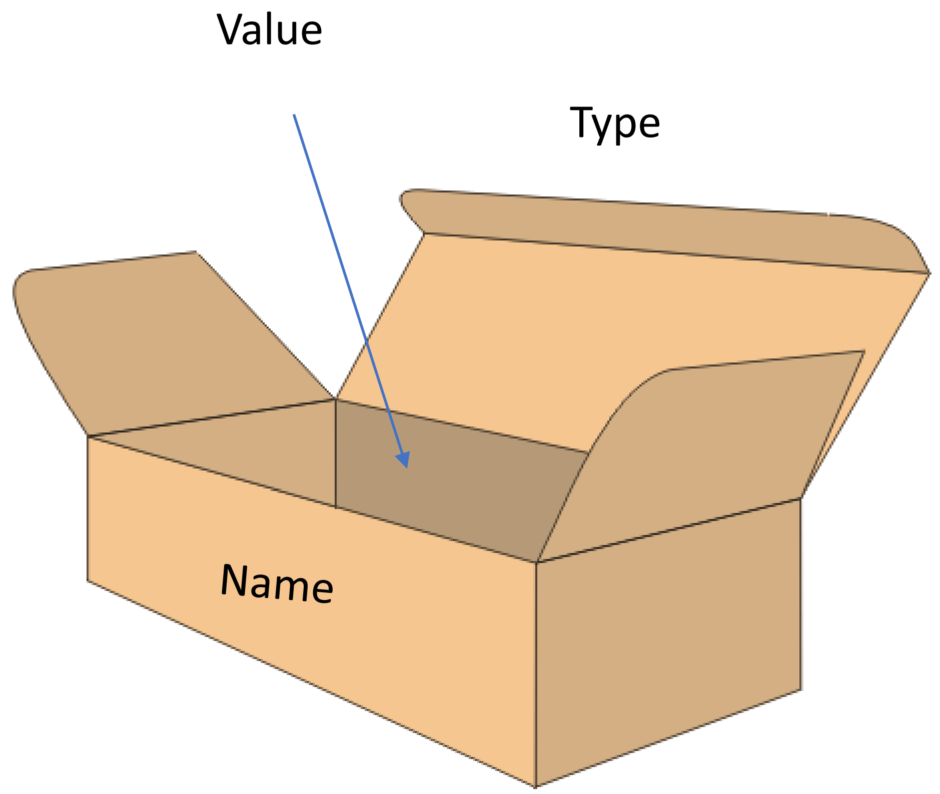
Multiple files:

```
g++ filename1.cpp filename2.cpp ...
```

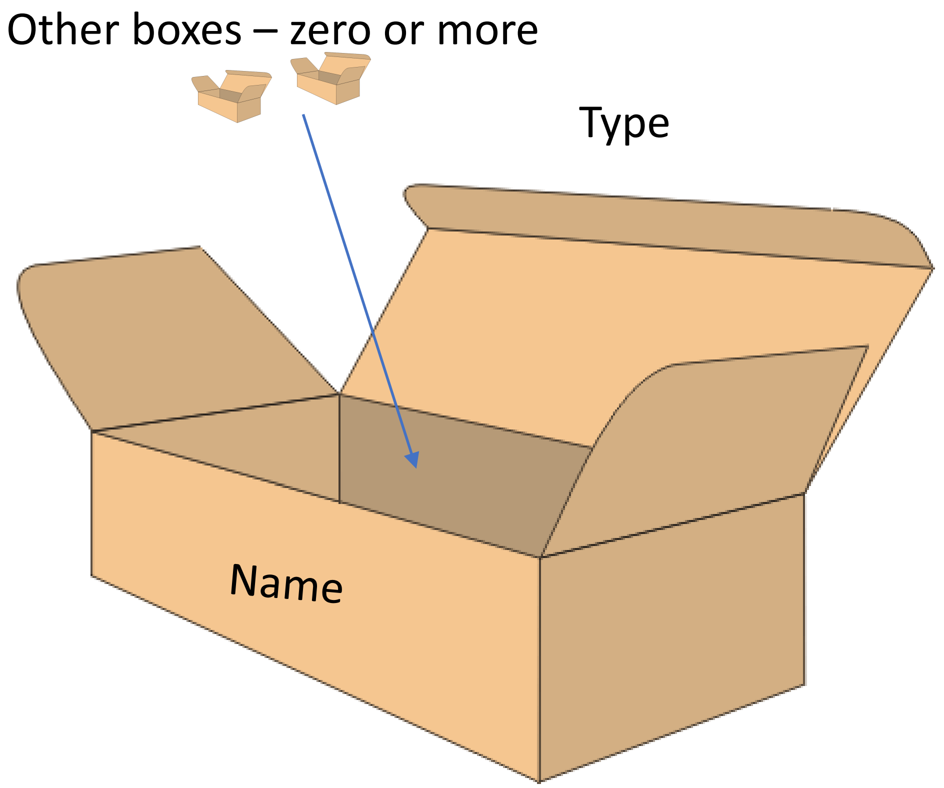
Variable

- Fundamental (also called built-in types)
  - Stores a value of a fundamental type, nothing more
- Object
  - Stores values tied to an derived type (struct, class)
  - Operations associated to the type are provided
  - More about classes later in the course
- Pointer – later in the course
  - Stores the address of some other variable
  - More about pointers in the course

Variable



Struct – Compound data type

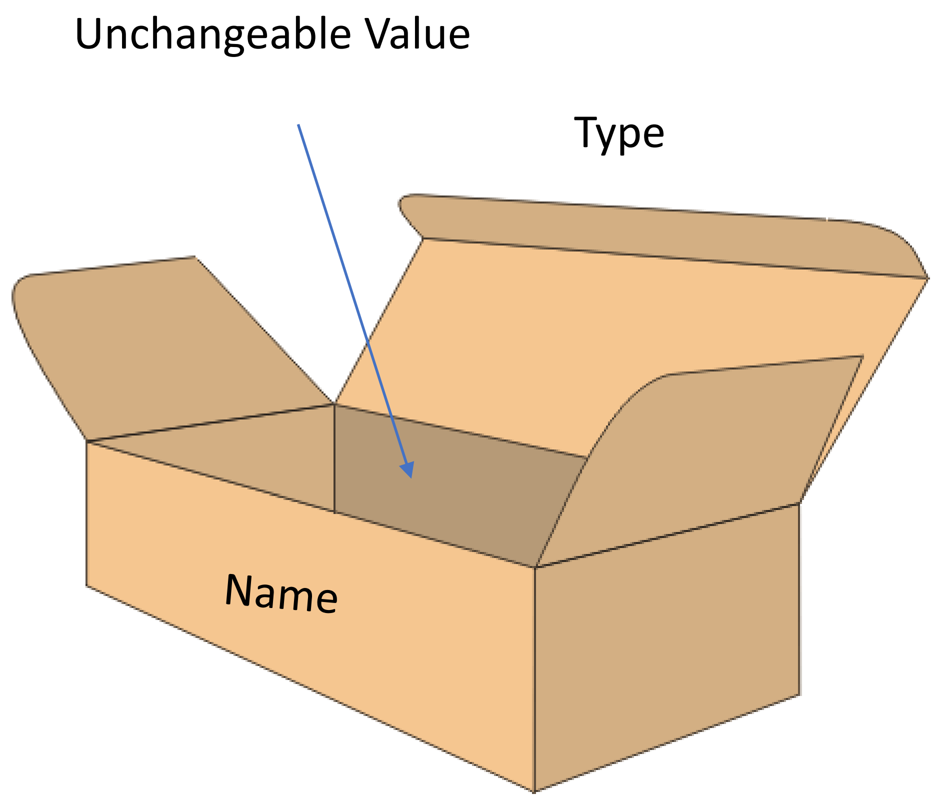


- With struct it is possible to combine variables into one derived type

Struct – Person

```
struct Person {
     string first_name;
     string last_name;
     int age;
};

Person p{“Sam”, “Le”, 32}; // Create a variable p of type
Person

cout << p.last_name << endl; // Will print out ‘Le’

p.age++; // change age to 33
```

Constants



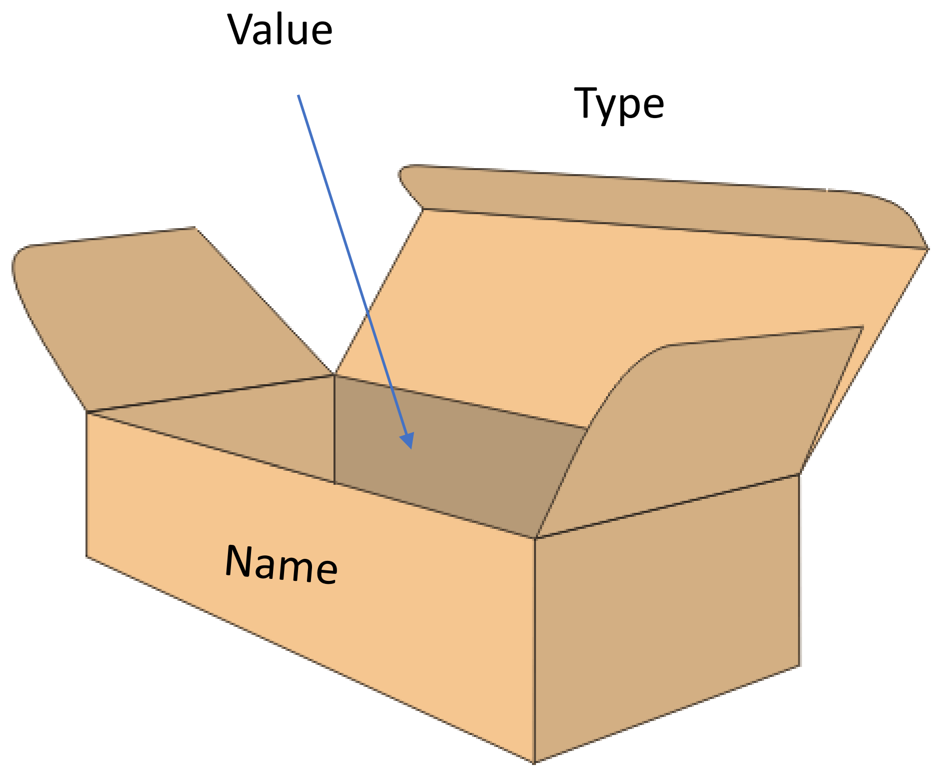
- A variable can be declared *const*
- Modification of a const variable will give compilation error.

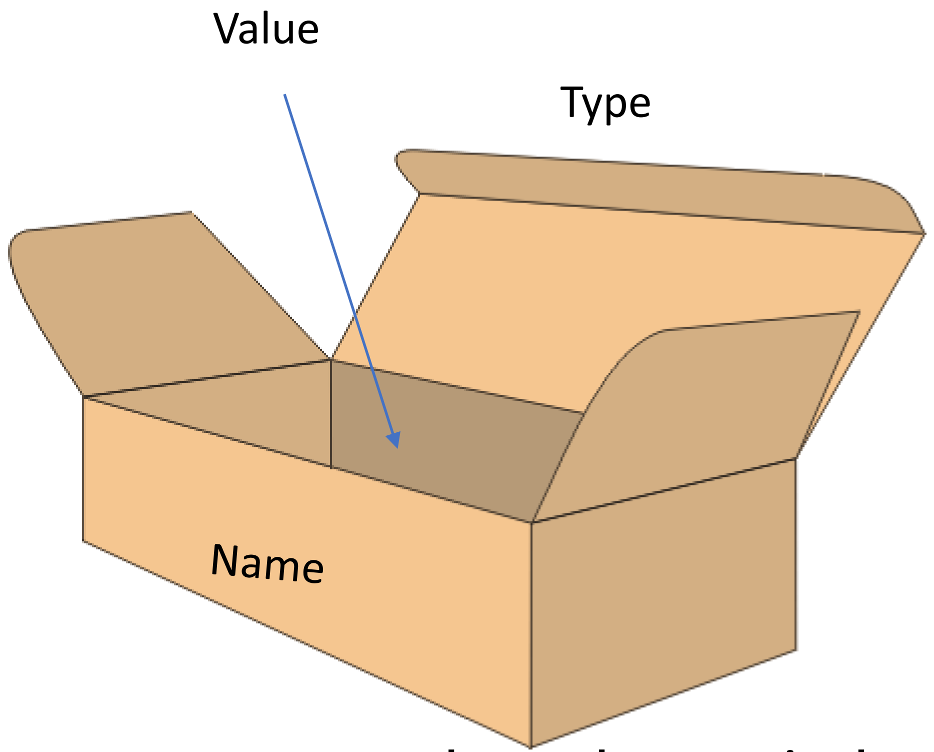
Constants - Example

```
int const x{3};
Person const sam{“Sam”, “Le”, 32};

x = 4; // Compilation error
sam.age++; // Compilation error
```

A good practice is to always use const when you can.

Copy



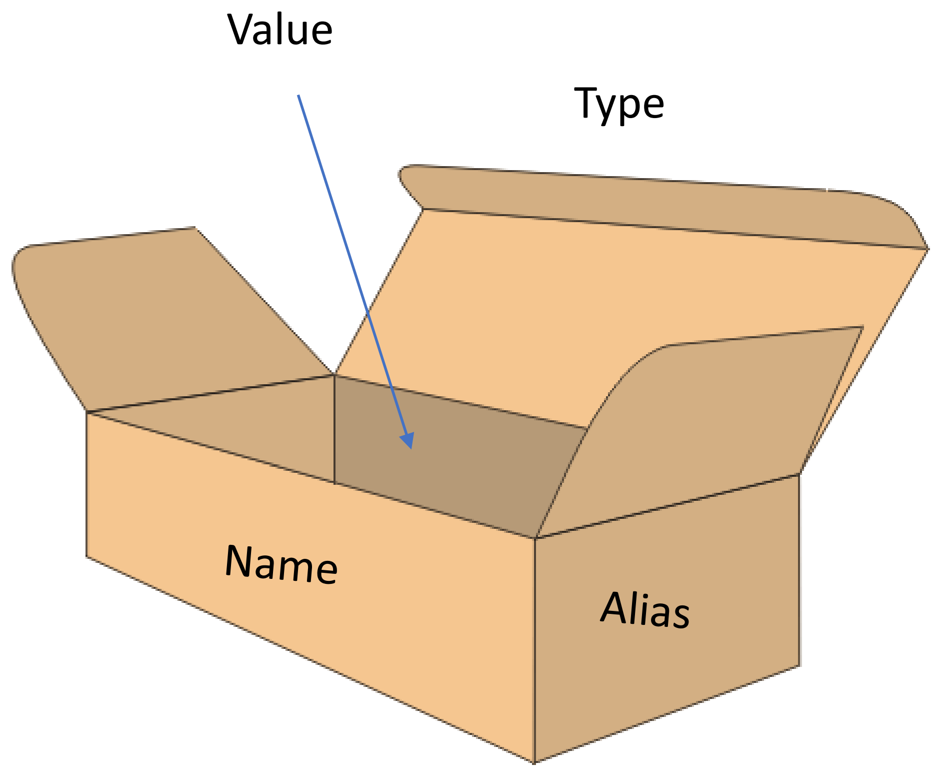


- When copy, the value and type will be an exact match to the copied value.
- A new variable is created, along with a new name.

Copy – Example

```
int a{5};
int b{a};

Person sam{”Sam”, “Le”, 33};
Person copied_sam{sam};
```

Reference



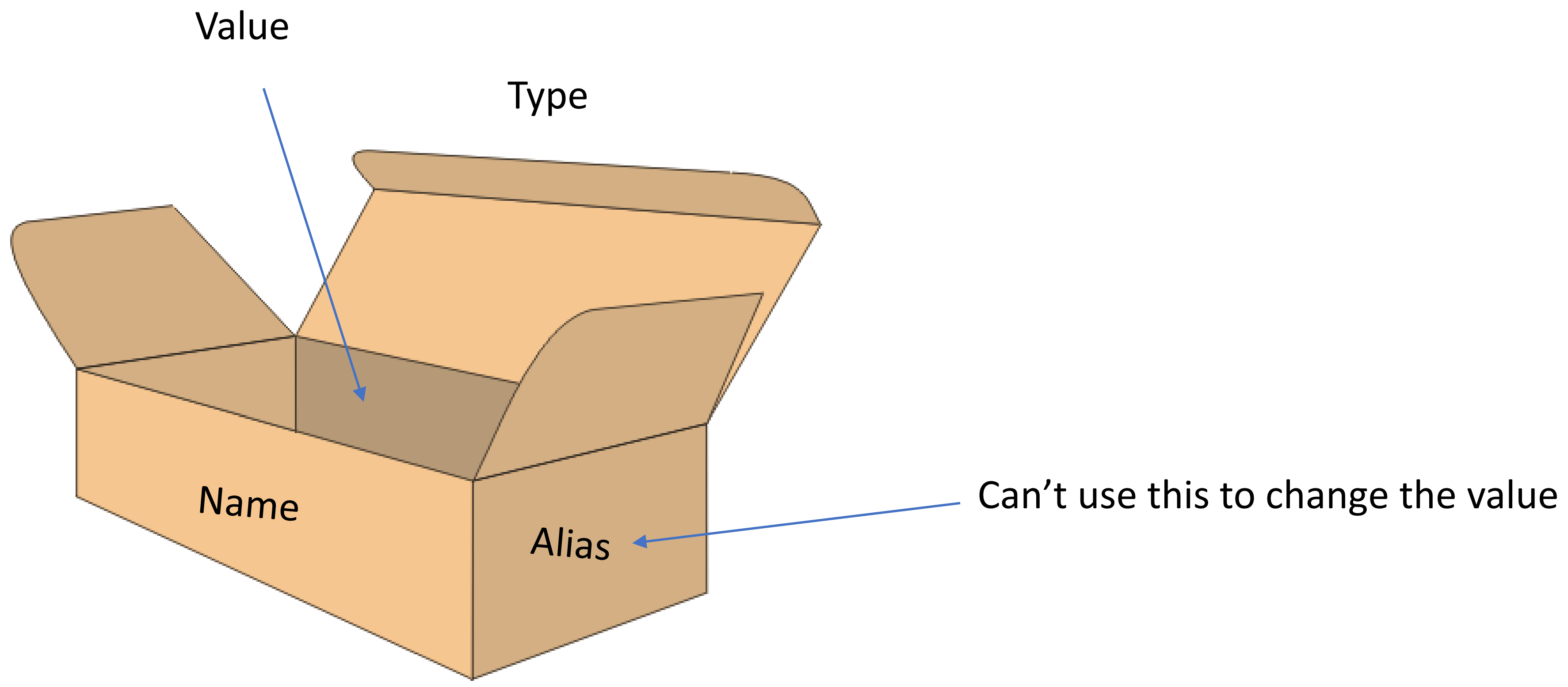
- Alias to another already existing variable
- A reference cannot refer to another variable after definition

Reference - Example

```
string professor{“C. Kessler”};
string & clever_fellow{professor};

clever_fellow = “F. Heintz”;
cout << professor << endl;
```

What will be printed?

Const&



- The value could be change using the original name but not the alias

Const& - example

```
Person sam{“Sam”, “Le”, 32};
Person const& also_sam{sam};

sam.age++; // Works perfectly fine
also_sam.age = 33; // Compilation error
```

Scope and block

- Each name that appears in a C++ program is only valid in some portion of the code called its *Scope.*
- Many different types of Scope.
  - global
  - block stope
  - function
  - class
  - etc.

Block scope

```
{        // Beginning of the block
         statement 1;
         statement 2;
         statement 3;
}        // End of the block
```

A variable declared inside a block is only visible inside that block

Block scope – example

```
int x{0};
{
     int x{1};
     {
          cout << x << ” “;
          int x {2};
          cout << x << “ “;
     }
     cout << x << ” “;
}
cout << x << endl;
```

Function types

- Global functions – Visible everywhere in you program after you declaration
- Member functions – A function that is a part of a class
- Lambda functions – A function created inline, or “on the fly”
- Function objects – An object possible to call as a function

Function

- A block that has been given a name
- Also called a subroutine or procedure if there are no return value.
- Visible after declaration
- Can be executed (called) by writing it’s name in other parts of the program

Function – basic syntax

```
return-type function-name(parameter-list) {
        statement1;
        statement2;
        return expression;
}
```

- **return-type** could be of any type that is in your program
- return **expression** must be of the return-type *declared*
- return statement exits the function

Function – examples

- A procedure

```
void foo() {
	cout << “the function foo” << endl;
}
```

- A function that add two integer and returns the value

```
int sum(int a, int b) {
	return a + b;
}
```

Function declaration and definition

- Declaration
  - Tells the compiler the function exists somewhere

  ```
  void foo();
  ```
- Definition
  - Places function code in program

  ```
  void foo() {
  }
  ```

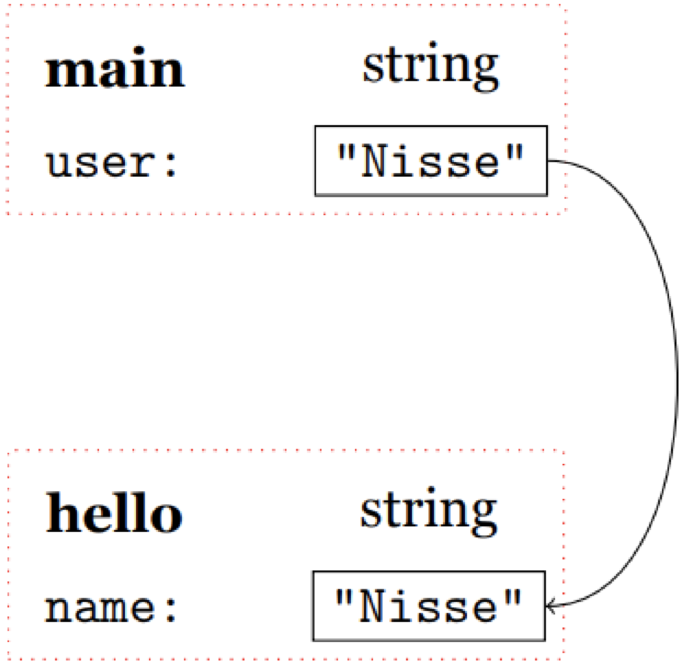
- Give the programmer a way to separate the program

Function declaration and definition

```
void hello();   // declaration

int main() {
    hello();
}

void hello() {   // definition
    cout << “hello” << endl;
}
```

Function – arguments

```
void hello(string name) {
        cout << “hello “ << name << endl;
}

int main() {
        string user{“Sam”};
        hello(user);
}
```



const and *reference* are applicable to arguments.

Function - Best practice for arguments

- Always use const in case fundamental types
  - int, double, char etc.
- Always use const& in case object types.
  - Person, string etc.

- Remove const only if you must modify the value

Function overload

- Different functions can have the same name
- Functions with same name must have different parameters
- Arguments given determine which function is actually called (closest match
- Compiler will select the “best match” among functions with the same name
- Return value is not considered even if different

Overloading example

```
int triangle_area(int base, int height);                   // version a
int triangle_area(int side1, int side2, int side3);        // version b
int triangle_area(int side1, int side2, double angle);     // version c
int triangle_area(int side, double angle1, double angle2);   // version d


triangle_area(1, 1);                          // calling version a
triangle_area(1, 1, 1);                       // calling version b
triangle_area(1, 1.0, 1.0);                   // calling version d
triangle_area(1, 1, 1.0);                     // calling version c
```

Default values

- Parameters can be given default values
- Specified in declaration only, since definition may be unknown to compiler if program is in several files
- Default values can only be specified for last non-default parameter
- Can be omitted when calling the function
- Combined with function overload then the declaration must be unambiguous!

Default values – example

```
void ignore(int n = 1, char stop = EOF);

int main() {
      ignore(); // call ignore(1, EOF)
      ignore(1024) // call ignore(1024, EOF);
      ignore(1024, ’\n’);
}

void ignore(int n, char stop) {
      cin.ignore(n, stop);
}
```

Operator overloading

- Define your own operators
- Customizes the C++ operators for operands of user-defined types

```
return_type operator symbol(left operand, right
operand);
```

Function – operators example

```
bool operator<(Person a, Person b) {
      return a.last_name < b.last_name;
}

int main() {
      Person sam{“Sam”, “Le”};
      Person cindy{“Cindy”, “Tran”};
      if (sam < cindy) {
            cout << “Sam’s last name comes before Cindy’s”;
      }
}
```

Function – operator example

```
ostream & operator<<(ostream & os, Person const& p) {
	os << p.last_name;
}

int main() {
	Person sam{“Sam”, “Le”};
	cout << sam;
}
```

File seperation

- Related functions can be gathered in one file to form a package.
- A package can be compiled separately, and do not need recompilation unless you change a package source file.
- Public declarations are place in a header file .h
- Definitions are placed in a implementation file .cc/.cpp
- Header and implementation files should have the same name, except for the extension

File separation – header file

```
// header file guard protect from multiple inclusion
#ifndef PERSON_H
#define PERSON_H
// DO NOT use namespaces here

// insert declarations here

#endif
```

File separation – implementation file

```
#include “person.h”

// definitions here
```

File separation – main

- In main you only include the header file. The rest will be handled by the compiler.

```
#include <iostream>
#include ”person.h”

int main() {
      Person sam{...};
      cout << sam;
}
```

Testing your program

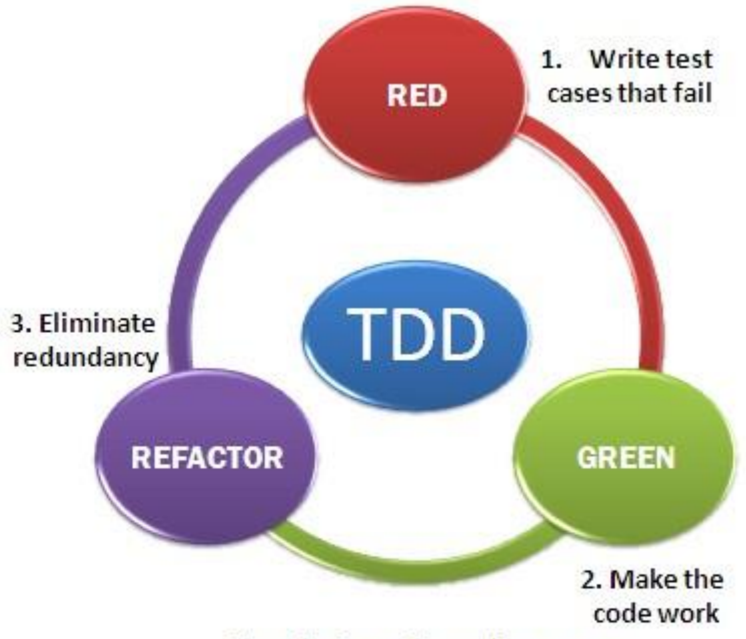
- So far you need to run you program and manually enter inputs to test if everything works as it should be
- This is tedious work and you could write a program to do this instead

```
int main() {
     doTestWithInput(1, 2, 3);
}
```

- This could be done but you need its not very intuitive to write
- The testing is done after the program is done

Test Driven Development

- Is a software development process
- Reverse the order of coding
  - Write test first
  - Implementation
  - Refactor code
  - Repeat
- The testing is done very regularly
  - When you write the test
  - When you implement
  - When you refactor
- You will immediately know when something break



Test Driven Development

Catch testing framework

- Catch is a simple testing framework that we will use in this course
- Its only one file that you include and everything will work
  - catch.hpp
- File separation when using catch (in this case lab2)
  - catch.hpp
  - test_main.cc
  - time_test.cc ← all the test cases
  - Time.h ← all the declarations
  - Time.cc ← all the implementations
- Compile everything with

```
g++ test_main.cc time_test.cc Time.cc
```

time_test.cc

```
#include “catch.hpp”
#include “Time.h”

TEST_CASE( “Test case name” )
{
     CHECK( condition );
     REQUIRE( condition );
}
```