C++ Essentials

C++ (and some C) for Ada (and Java) Programmers

Contents

1 Audience, C++ Background, Simple program

This guide introduces C++ to those who have seen procedural programming in Ada. Information on C and some comparison with Java is also provided.

1.1 C++ Background

- C++ is C with objects, created around 1985
- stronger type checking than C
  - Increasing type safety: C < C++ < Java < Ada.
- better parameter passing than C (ie references, like in out parameters)
- templates (ie generics) added in 1998
- Allows, but does not require, objects, like Ada
- Lots of braces, like C and Java

1.2 Output

Output from many cout (ie standard output) commands is given. Sometimes spaces are added to make the output more readable.

1.3 A Simple Program: table.cpp

This table prints a table of squares and cubes, up to a limit that the user enters. The limit must be positive. The sum of 1 up to the limit is also printed.

```
1  // File: table.cpp
2  #include <iostream>
3  #include <cmath>
4  #include <stdio.h>
5
6  using std::cout; using std::endl; using std::cin;
7
8  int square(int i){return i * i;}
9  int cube(int i);
10
11 void title(){cout << "A table of sums, squares, and cubes\n";}
12
13 int main()
14 {
15    int limit;
16    while(true)
17    {
18       cout << "Enter a positive limit: ";
19       cin >> limit;
20
21       if (limit <= 0)
22          cout << limit << " is not positive!" << endl;
23       else
24       {
25          std::cout << "Limit " << limit << " is valid" << std::endl;
26          break;
27       }
28    }
29
30    title();
31    int sum = 0;
32    for (int i=1; i <= limit; i++)
33    {
34       sum += i;
35       int sql = square(i);
36       int cubel = cube(i);
37       cout << i << sql << cubel << endl;
38       printf("%d\t%d\t%d\n", i, sql, cubel);
39    }
40    printf("The sum is %d!!\n", sum);
41 }
42
43 int cube(int i){return std::pow(i, 3);}
```

- C++, like C and Java, is case sensitive (eg `include` and `Include` are different identifiers)
- // - single line comment
  - `/* ... */` - multiline comment
- 1: **table.cpp -** file name can be anything.
  - compile with "`g++ table.cpp`" and run with "`./a`" (the default executable is named `a.out` on unix and `a.exe` on windows)
- 2, 3, 4: **#include <iostream>** , **#include <cmath>, #include <stdio.h>** - makes libraries iostream, cmath, and stdio accessible. Similar to Ada with statements
  - **No semicolon!!!!**
  - **#include <iostream>**: This line is somewhat similar in function to an Ada `With` statement. Lines that begin with # are known as preprocessor directives, and to handle this statement the preprocessor finds the header file (which is similar to an Ada `.ads` file) associated with the library iostream and includes that at the location of the #include. This makes the declarations for all of the functions in library iostream available to the compiler, just as if they had been in the file at that location.
  - iostream is a C++, not a C library.
  - math and stdio are libraries made available from C. Since these are C libraries, they can be included using either the C++ convention (ie `cmath` and `cstdio`) or the C convention (`math.h` and `stdio.h`). One of each is included in this example.
  - See section **Preprocessor**
- 6: **using std::cout; using std:endl; using std::cin** - These statements are somewhat similar to Java `import` and Ada `use` statements. They make the functions `cout`, `end`, and `cin` available without needing to give the namespace (ie `std`). Without these statements, the function `cout`, for example, would need to be called using its namespace, as in `std::cout`, but with these using statements, `cout` can be called using just its name (ie `cout` rather than `std::cout`).
  - `cout` is the output stream for standard output and `endl` is a newline. Both are defined in library iostream. More detail is given below when the call to `cout` is described.
  - `using std:cout, std:endl, std:cin;` is available in later versions of C++ (ie one `using` statement with a comma separated list of functions)
  - A `using` statement can be put inside a function, making it local to that function only, as with Ada `use` statements.
  - Could also say **using namespace std;**, but this would make every name in namespace `std` available, but namespace `std` includes **every** member of C++'s Standard Library, making it more like to have name clashes with user names.
  - A function can be called with a fully qualified name (eg `std::cout`), even if there is a using statement. Examples, such as **std::cout** occur later in the code.
- 8: Declares and defines function **square**. Notice that the function `square` is NOT nested in any other function. This is different from Ada: More than one non-nested function can be declared in a file, that is, a file can contain more than one function declared at the top level (ie not nested in any other function). To save space, the function is defined on a single line.

- 9: Declaration (ie signature) for function **cube**, which is defined below, after main. Like Ada, a call to a function must have a declaration ot that function above it (possibly via a `#include`).
- 11: Void function **title** is like an Ada procedure. It has no parameters, but () is required
- 11: **cout -** an output stream defined in library iostream (as noted above)
  - `cout` sends output to **Standard Output**
  - **cout << "string\n"** - concatenates string and a newline to `cout` (ie to Standard Output)
- 13: **main** must return an int and must have exactly either 0 or 2 parameters
  - Two parameters for argument count and an array of command line argument strings
- **16: while (true)** - like Java, if and loop conditions are in parens.
  - C++ has built-in constants **true** and **false** of type **bool,** but these are easily **converted to integer types (eg bool b = 33+44; int i = true * false;)**
  - Like C, C++ has no Boolean type: 0 is evaluated as false, others values as true. A library stdbool defines type bool and constants true and false, but the type is an integer type and the constants are integers.
- 18: Notice no newline for the prompt
- 19: `cin`, defined in iostream, is the input stream connected to Standard Input
  - >> `limit` puts the next integer that is in standard input into variable limit
  - This loop is infinite if `cin` encountres a non-integer
- 22: Adds several strings to hte output stream
  - `limit` - value of `limit` is converted into a string for output
  - **endl** - adds a newline. Defined in iostream. (As noted above)
  - Single statement if (and loop) bodies needs no braces, Like C and Java.
- 25: **std::cout, std::endl -** shows that you can explicitly give the namespace containing `cout` and `endl`. Without the **using** statement, this explicit namespace would be required.
  - All members of the C++ Standard Library are automatically included in the namespace **std** (which means standard). The Standard Library includes I/O, containers, threads, math etc, and so there are lots of members!
- 26: **break** the loop, as in C and Java. Equivalent to `exit` in Ada.
- 30: Call void function (ie procedure) named `title`
- 31: In early version of C, declarations had to be at the beginning, like Ada, but this has been relaxed.
- 32: Loop looks just like Java. This loop starts at 1 to make the table begins at 1
- 34: **+=:** Yes, you can save a few symbols of typing
- 37: This line prints the values, without any spacing. Although the line produces redundant, duplicate output, it's included here as another example of `cout`.

- 38: **printf** - formatted print. `Printf` is from C. `Cout` is not available in C.
  - `Printf` prints the string that is its first parameter, but first it makes any substitutions that are specified in the string. Substitutions are done for each format specifier. The format specifier `%d` is for substituting and formatting a signed integer value. The remaing parameters to `printf` are subsituted in the string in order. So the three `%d` are replaced by the value of `i, sql, cubel`, in that order.
  - The escape seqence `\t` represents a tab character.
  - `%d` is for signed integers. Other format specifiers include `%f` for floating point, `%s` for string, and `%x` for hexadecimal. Do a search for more specifiers.
  - Printf does not type check its substitutions, and so you can print a float as an int and vice versa!
  - Printf is from C library `stdio.h`.
- 40: **printf** - prints the sum as part of a sentence.
- 43: Definition of function **cube,** whose declaration is given above `main`.
  - **std::pow -** Calls **pow** from cmath from the Standard Library. Since `cmath` is part of the Standard Library, all of its members are included in the namespace `std`, and since there is no statement: `using std::pow`, the namespace `std` must be given explicitly.

1.4 Compilation, file extensions, and execution

This document was tested with the C++ compiler g++ version 8.3.1 on Windows 10, which is part of the gcc family. Some code was also compiled on linux.

Warning: On rucs, there are several different C++, and link errors can occur if the path is not set correctly. Default bash paths for students should work correctly. The compilers on rucs are at `/usr/bin/g++`, `/usr/local/bin/g++`, `/usr/local/gnat/bin/g++`.

For source file table.cpp:

- Unix:
  - Compile: `g++ table.cpp`
    - Creates **executable** `a.out`.
  - Run: `./a`
- Windows:
  - Compile: `g++ table.cpp`.
    - Creates executable `a.exe`.
  - Run: `./a`
- Do I have a C++ compiler on my Windows machine? Yes, if you have installed the gnat Ada compiler. Gnat's bin directory (typically `C:\Gnat\bin`, the same directory where gnatmake is) should contain the C++ compiler, g++, and the C compiler gcc.
- Specify the executable name (eg table) with -o:
  - `g++ -o table table.cpp`
- Common extensions:
  - C++ programs: `.cpp`, `.C`, `.cc`, `.c++`, and `.cp`
  - C programs: `.c`
  - Header files:
    - Header files, which contain function declarations (ie signatures), are used when breaking a program into multiple files
    - C headers: `.h`
    - C++ headers: `.h` and `.hpp` are common

2 Functions (and procedures) and parameters

2.1 Functions: void and non-void (ie procedures and functions)

- A C++ function is similar to a Java method.
  - An Ada function is similar to a C++ function with a non-void return type.
  - An Ada procedure is similar to a C++ function with a return type of void.

2.2 Functions and Procedures: Never the Twain Shall Meet?

- In Ada, a function can never be used in place of a procedure and vice versa
  - A function represents a value and is used only as an expression (or part of one)
  - A procedure is used only as a statement
- In C and C++, a function's return value can be ignored (so a function with a non-void return type can be called like a procedure.
  - For example, the following fragment, which uses `sqrt` as both a value and a statement, should compile:

```
1 float f = sqrt(2.0);
2 sqrt(2.0);    // Ignore return value
3 f - 1;        // Ignore result of the expression (ie do nothing!)
```

2.3 Functions: Declare and define (and where)

- Terminology: **Declaring** a function is giving its signature; **defining** a function is giving its signature and its body
  - This same distinction holds for variables, and other things, in addition to functions
- A C++ function must be declared above where it is called. Several options are possible for declaring and defining a function:
  - A function f can be defined above a function that calls f
  - A function f can be declared above a function that calls f, and f can be defined after the function that calls f
  - A function f can be defined in a separate file and a declaration (ie signature) of that function can be placed in a .h file, and that .h file can be used in a #include directive in the file of a program that calls f (making the signatures in the .h file available).
  - Function f can can be defined in a library and a `#include` can be used for that library (similar to the previous option)

2.4 Parameters in C++

- Default mode: like Ada `in` mode.
  - Unlike Ada's `in` mode, the formal parameters can be changed in the called function
  - But, like Java, a change in a formal parameter does not change the actual parameter
- Can also have the equivalent of `in out` mode:
  - Changes made in the formal routine are reflected in the actual parameters

```
1  // File: paramModes.cpp
2  #include <iostream>
3  using std::cout; using std::endl;
4
5  void doesNotSwap(int i, int j){  // Does not swap actual parameters
6     int t = i;
7         i = j;
8         j = t;
9  }
10
11 void swap(int &i, int &j){        // Swaps actual parameters
12    int t = i;
13        i = j;
14        j = t;
15 }
16
17 int main()
18 {
19    int x=11;
20    int y=22;
21
22    doesNotSwap(x, y);
23    cout << x << y;      //  Output: 11 22
24
25    swap(x, y);
26    cout << x << y;      //  Output: 22 11
27 }
```

- 11: The `&` in the formal parameter list means that the parameter is passed **by reference**
  - The address of the actual parameter is passed to the function
  - Changes made in the function are reflected in the caller
  - Similar to an Ada `in out` mode parameter
  - We will also use & later when talking about pointers
- Without the &, a copy of the actual is passed to the function (ie passed **by value)**
  - A copy of the parameter is copied in to the function. No value is copied back
  - Similar to an Ada `in` mode parameter
- Arrays are automatically passed by reference, as discussed later

2.5 Parameters in C

- Default is same as C++ (and Java) and similar to Ada `in` mode parameters
- To get something like C++ reference parameters (ie like & parameters, similar to Ada `in out`), the user must explicitly specify passing the address of the actual

```
1  // File: paramModes.c
2  // This is C, not C++
3
4  #include <cstdio>
5
6  void doesNotSwap(int i, int j){      // Does not swap, like C++
7          int t = i;
8              i = j;
9          j = t;
10 }
11                                      // This version swaps
12 void swap(int* ip, int* jp){        //  i and j are pointers to ints
13     int  t = *ip;                    // Explicit dereference
14         *ip = *jp;
15         *jp =  t;
16 }
17
18 int main()
19 {
20     int x=11;
21     int y=22;
22
23     doesNotSwap(x, y);
24     printf("%d %d", x, y);      //  Output: 11 22
25
26     //  Explicitly pass addresses of x and y
27     swap(&x, &y);
28     printf("%d %d", x, y);      //  Output: 22 11
29 }
```

- 12: `int* ip // ip is a pointer to an int`
  - Can use several spacings, all equivalent: `int* ip; int * jp; int *kp;`
- 13: `t = *ip // t is assigned the int at which ip points`
  - `*ip // dereference the pointer ip`
- In C++ the actual parameter's address is passed implicitly and in the routine that address is dereferenced implicitly in the called routine
- In C, the actual parameter's address is passed explicitly and that address is dereferenced explicitly in the called routine

3 Header files, Breaking a program into separate files, and Linking

It is common to define a set of helper routines that are called from a main routine. In Ada the helper routines can be defined in a package: the `.ads` file has the specification of what is available in the package, and the corresponding `.adb` file provides the implementation of that package. In C++, a `.h` file contains the function declarations (ie signatures) of the helper routines, and a corresponding `.cpp` file contains the definitions (ie implementation) of those routines. The `.h` file can also contain types that are needed by the helper and by the main routine.

```
1  ////////////////////////////////////
2  // File: testmath.cpp
3  #include "myMath.h"
4
5  #include <iostream>
6  using std::cout; using std::endl;
7
8  int main(){
9     int limit = 3;
10    for (int i=1; i<= limit; i++)   {
11       int sql = square(i);
12       int cubel = cube(i);
13       cout << i << sql << cubel << endl;
14    }
15 }
16 //  End of test.cpp  ////////////////
17
18
19 ////////////////////////////////////
20 // File: myMath.h
21 // Signatures only
22 int square(int i);
23 int cube(int i);
24 //  End of myMath.h  ////////////////
25
26
27 ////////////////////////////////////
28 // File: myMath.cpp
29 #include "myMath.h"
30 #include <cmath>
31
32 int square(int i){return i * i;}
33 int cube(int i){return std::pow(i, 3);}
34 //  End of myMath.cpp  ////////////////
```

- 4: When including a **local file** (vs a library file), **use quotes rather than <>**
- 21: The file extension `.hpp` is also used for C++ header files. Some tools require specific extensions.

- **Compilation: Compile and link to create executable, all in one command:**
  - `g++ myMath.cpp testmath.cpp`
  - Other ways of compiling are possible:
    - `g++ -c` means to compile only, with no linking and not creating an executable
    - Compile the library (create `myMath.o`): `g++ -c myMath.cpp`
    - Compile (but don't link) the main (create `testmath.o`): `g++ -c testmath.cpp`
    - Can pass `.o` file(s) to g++:
      - `g++ myMath.o testmath.o`
      - `g++ myMath.o testmath.cpp`
- To have myMath compiled automatically, you would have to create a **make file**
  - A make file specifies the dependencies needed to do correct compilation, and the make program does the compilations required to create the executable
  - **gnatmake** does what make does (ie compile needed files automatically)
- Warning: Avoid compiling the header. Don't do this!!: `g++ myMath.h`
  - This creates `myMath.h.gch` file which is cached version of the `myMath.h`, but this file is NOT automatically updated if the `.h` file is changed (unless the `.h` is explicitly recompiled)

3.1 Linking and Link (and Load) errors

- The compiler g++ compiles all `.cpp` files and creates (and then deletes) corresponding `.o` files .
  - In these `.o` files, there may be places that should contain calls to functions that are defined in another file (ie calls to a library routines or to user-defined helper functions). Since the compiler does not know where these routines are, the destination of the calls are left blank, to be completed at the later, at the link step
- After all `.o` files are created, the linker creates the executable by doing the following:
  - Combining the `.o` files for all user-defined routines as well as all needed library routines
  - Replacing the holes in the `.o` files with the destination of the calls
    - This can be done once all of the needed `.o` files have been combined and the call destination locations are known
- Truth in advertising: the simplified description above ignores static and dynamic libraries (eg `.a` archive files)
- Link errors: Files that compile correctly can still have errors when linked together:
  - Forgetting to list all files for the compilation (eg `g++ foo.c` instead of `g++ foo.c mylib.c`)
  - Function declared in the header are not declared (or are declared incorrectly!) in the cpp file.
    - Yes, the check for consistency between `foo.h` and `foo.cpp` occurs at link time!
  - Not having a guard (see Preprocessor section) can cause there to be multiple declarations of a variable
  - Environment not set up correctly so that the linker can't find library files
    - The C/C++ linker is called `ld`, and so `ld` shows up in the message for linker errors

4 Preprocessor

C and C++ compiles in several steps. First the preprocessor transforms a source program into a translation unit, and the following steps (eg compile and link) operate on this translation unit. A program contains preprocessor directives that cause the preprocessor to make textual substitutions. Directives begin with a # and each is on a line by itself. The line does not end with a semicolon. After preprocessing, the directives are gone from the translation unit. Common directives include #include, #define, #ifndef, #endif.

- **#include**: Brings the specified file into the translation unit. The file is normally either a user defined header or a Standard Library header. For example, `#include <iostream>` directs the preprocessor to add the iostream header and `#include ‘‘myLib.h’’` add user-defined header `myLib.h` to be added. A header normally contains types and function signatures.

- **#define**: Defines a macro that is expanded when used. A macro applies from the point of definition and below, but not above. Here is an example:

```
1 #define PI 3.14149
2 #define FINDMAX(a, b) a>b?a:b
3
4 double d = 2 * PI;   // Macro expands to d = 2 * 3.14159;
5 v = FINDMAX(x,y);    // Macro expands to v = x>y?x:y
```

  - The preprocessor expands `PI` to 3.14159 everywhere (below its definition) that `PI` occurs in the source text. After the preprocessor step, the macro `PI` is gone, and only the 3.14149 remains. This is different from defining `PI` to be a constant (`eg const double PI=3.14159;`)
  - The 2 parameters of macro `findmax` are substituted, before compilation, when findmax is used.

- **#define** and **#ifndef - guards**: These directives guard against a file being #included more than once.

  - If file A #includes B and C, and B and C #include D, then D will be #included twice, which might cause problems. This can also occur if C includes D indirectly, since if A #includes C and C #includes D, then A will effectively #include D (ie include is transitive).
  - To avoid multiple #includes, a guard (eg `FOO_H_INCLUDED)` detects that a file has already been #included.

```
1 // File: foo.h
2 #ifndef FOO_H_INCLUDED
3   #define FOO_H_INCLUDED
4   #define PI 3.14149
5   #define findmax(a, b) a>b?a:b
6 #endif
```

    When foo.h is included in file A, then the definitions of `PI` and `FINDMAX` will only be added to A if `FOO_H_INCLUDED` is not defined (ie #ifndef). Also, if `FOO_H_INCLUDED` has not yet been defined, then it will be defined. Notice that `#define FOO_H_DEFINED` does not provide a value; it simply says that it's defined so that a subsequent check will find it already defined.

The preprocessor can also be used to create **different code for different platforms.**

5 Nesting, Local, Top-level, Global, Visibility, Scope

In Ada, a variable can be declared inside a subroutine (a local variable) or inside a package (and not inside a subroutine). In Java, a variable can be declared inside a method (a local variable) or inside class (and not inside a method).

In C++ (and C) a variable can be declared in a function (a local variable). Other places that a variable can be declared include outside a function (ie in the same file as the function) or in a header (ie `.h`) file.

In Ada, subroutines can be nested inside other subroutines. In Java, methods are not normally nested in other methods (well, anonymous inner classes and lambdas are nested). In C/C++ a function can not be nested within another function.

In Ada there is one top-level (ie non-nested) subroutine per file (ignoring packages). Java is similar with top-level classes. In C/C++, a file can contain more than one top-level function and one or more variable that is declared (or defined) at the top level (ie not local).

5.1 Nesting

- Functions can not be nested in other functions.
  - (In contrast, Ada subroutines are commonly nested)

5.2 Local, top-level, global

- Variables declared within a function are **local variables.**
- Variables and functions that are NOT declared in a function are **top-level variables and functions**
  - A file can contain more than one function at the top level
    - This is different from Ada which allows only one top level subroutine in a file
- Functions and variables declared at the top level are **global**.

5.3 Visibility and Scope

- Global (ie top level) variables and functions are visible:
  - Within the file, from the point of declaration on (but not above the declaration)
  - To any other file
    - But their visibility can be controlled with **namespaces.**
    - To be visible in another file, a global variable must be declared as **extern** in the other file
    - Global variables can be declared in a `.h` or a `.cpp` file, but declaring them in a `.cpp` file is normally not good practice
- Local variables are visible from the declaration to the closing brace of the closest enclosing braces. This is essentially the same scope rule as Ada and Java.
- The scope of a local variable is from the declaration to the corresponding closing brace
- The scope of a global is the entire program (ie all files being compiled), as modified by the **namespace**

6 Namespaces, Default Namespace std, Standard Library

- **Namespaces:**
  - Global functions and variables can be declared within **namespaces:**
  - Namespaces provide a means to control access to global names
  - Each global function or variable is in a namespace, either in the default namespace or nested within a user-defined namespace.

```
1  // File: testmath.cpp
2  //  Shows use of namespace mathnames
3
4  #include "myMath.h"
5
6  #include <iostream>
7  using std::cout; using std::endl;
8
9  int main(){
10    for (int i=1; i<= 5; i++)   {
11       int sql = mathnames::square(i);       // Use namespace mathnames
12       int cubel = mathnames::cube(i);
13       cout << i << sql << cubel << endl;
14    }
15 }
16
17 ////////////////////////////////////
18 // File: myMath.h
19 namespace mathnames  // Add signatures to namespace mathnames
20 {
21    int square(int i);
22    int cube(int i);
23 }
24
25 ////////////////////////////////////
26 // File: myMath.cpp
27 #include "myMath.h"
28 #include <cmath>
29 namespace mathnames    // Add bodies to namespace mathnames
30 {
31    int square(int i){return i * i;}
32    int cube(int i){return std::pow(i, 3);}
33 }
```

- Namespace `mathnames` includes the signatures in `myMath.h` and the definitions in `myMath.cpp`.
- The names `mathnames::square` and `mathnames::cube` explicitly specify the namespace in which to find `square` and `cube`

A **using statement** would allow direct access to the names in the namespace `mathnames`:

```
1  //  Shows use of namespace mathnames
2
3  #include "myMath.h"
4  using namespace mathnames;  //  Make all mathnames members directly accessible
5
6  #include <iostream>
7  using std::cout; using std::endl; // Make cout and endl directly accessible
8
9  int main(){
10    for (int i=1; i<= 5; i++)   {
11       int sql = square(i);   // Direct access to mathnames::square
12       int cubel = cube(i);
13       cout << i << sql << cubel << endl;
14    }
15 }
```

6.1 Default namespace

- Global functions and variables are in the default namespace (eg `::foo`) if no other namespace is specified

```
1  //  Shows use of default namespace
2  #include <iostream>
3  using std::cout; using std::endl;
4
5  #include "myMath.h"
6
7  int main(){
8     for (int i=1; i<= 5; i++)   {
9        int sql = ::square(i);   //  Specify the default namespace
10       int cubel = ::cube(i);
11       cout << i << sql << cubel << endl;
12    }
13 }
14
15 ////////////////////////////////////
16 // File: myMath.h
17 //  These are in the default namespace
18    int square(int i);
19    int cube(int i);
20
21 ////////////////////////////////////
22 // File: myMath.cpp
23 //  These are in the default namespace
24 #include "myMath.h"
25    int square(int i){return i * i;}
26    int cube(int i){return i * i * i;}
```

7 C++ Standard Library, Namespace std, C Standard Library

- The Standard Library contains functions and classes for I/O, math, strings, exceptions, containers, threads, and others.
- Make declarations in the Standard Library available by including the proper header file (eg `#include <iostream>` or `#include <exception>`)

7.1 Namespace std and the Standard Library

All names in the C++ Standard Library are declared within the **namespace std** (for standard) Like other namespaces, declarations in the Standard Library can be accessed in namespace std in 3 ways:

1. explicit: `std::cout ...;`
2. use a member: `using std::cout;`
3. use all members: `using namespace std;`
    - Making all names in `std` available, although convenient, can lead to name clashes and errors

Examples

```
1  //Explicit:
2  #include <iostream>
3
4  int main(){std::cout << "hi" << endl; }
5
6  //Use a member:
7  #include <iostream>
8  using std::cout; using std::endl;
9
10 int main(){cout << "hi" << endl;}
11
12 //Use all member:
13 #include <iostream>
14 using namespace std;
15
16 int main(){cout << "hi" << endl;}
```

7.2 C Standard Library

- **Backward compatibility**: **Each C Standard Library** header file is available, with a new names as well as its original name. The new name adds c at the beginning and removes .h. For example, the original name `stdio.h` is replaced with C++ library file cstdio. Thus, a C++ program can use either of the following:
    - `#include <stdio.h>`
    - `#include <cstdio>`

8 Types, type checking, and implicit conversions

Built-in types are shown below. These are all primitive types (ie values, not references). Like Java, char is an integer type.

Basic type	Short/Long	Unsigned	Signed	Bits	Bytes
`char`		`unsigned char`	`signed char`	8	1
`int`		`unsigned int`	`signed int`	32	4
	`short int`	`unsigned short int`	`signed short int`	16	2
	`long int`	`unsigned long int`	`signed long int`	32	4
	`long long int`	`unsigned long long int`	`signed long long int`	64	8
`float`				32	4
`double`				64	8
	`long double`			128	16
`wchar_t`				16	2

- Sizes in bits and bytes are shown, using gcc 8.3.1 on 64-bit processor
- The usual ranges for n bits:
  - signed:2^{n-1}.. $2^{n-1}-1$
  - unsigned: 0 .. 2^n-1
- C and C++ do not specify the sizes of various integer types, but they do specify minimum ranges.
  - This is like Ada, but different from Java, which defines the size of each type.

9 Arrays

9.1 Introduction to Arrays

Array declarations look somewhat like Java

```
1  //Arrays
2  #include <iostream>
3
4  int main(){
5    int size = 5;
6
7     //  int arr1 [];                              // Error: Size not known
8     int arr2 [size];                              // Size 5
9     int arr3 [size] = {11, 12, 13, 14, 15};       // Size 5
10    int arr4 [    ] = {11, 12, 13, 14, 15};       // Size 5
11    int arr5 [size]{11, 12, 13, 14, 15};          // Size 5
12    int arr6 [   9] = {11, 12, 13, 14, 15};       // Size 9
13    // int arr7 [2] = {16, 2, 77, 40, 12071};     // Error: too many initializers
14
15    int sum = 0;
16    for (int i = 0; i < size; i++)
17       sum += arr3[i];
18    std::cout << sum << std::endl;    // 65
19 }
```

- 11: Initializer for `arr5`
- 12: last 4 initial values of `arr6` are 0
- Indices start at 0, access with square brackets (As in Java)
- These arrays are allocated on the stack. Heap allocated arrays are shown in the section on pointers

9.2 Array Bounds are Not Checked

A program can access elements outside the bounds of an array

```
1  //Array parameter sizes are not checked
2  #include <iostream>
3  using std::cout; using std::endl
4
5  int main(){
6     int x = 55;
7     int arr[2] = {91, 92};
8     cout<< arr[-1]<< arr[0]<< arr[1]<< arr[2]<< arr[3]<< endl;
9     //          9725142   91        92        55     16782736
10 }
```

- 8: `a[-1]` accesses the value that happens to be in memory before the array (ie 9725142)
- 8: `a[2]` accesses the value of `x`, the variable stored after the array
  - Does this look backward? This is because the variables are allocated on the stack in reverse order (ie essentially `arr` before `x`)
- 9: 16782736 is the value that happens to be in location `a[3]`
  - the value that happens to be at `a[0]` and `a[3]` can be different on each run of the program

9.3 Size Also Not Checked for Arrays as Parameters

- Not surprisingly, array parameters are not checked for size

```
1  //Array parameter sizes are not checked
2  #include <iostream>
3  using std::cout; using std::endl
4  int sumFirstThree(int a[100]){
5     return a[0] + a[1] + a[2];
6  };
7
8  int main(){
9     int arr1[6] = {11, 12, 13, 14, 15, 16};
10    cout  << sumFirstThree(arr1) << endl;     // 36
11
12    int arr2[3] = {21, 22, 23};
13    cout  << sumFirstThree(arr2) << endl;     // 66
14
15    int arr3[3] = {91, 92};
16    cout  << sumFirstThree(arr3) << endl;     // 204 = 91+92+21
17 }
```

9.4 Size frequently passed as a parameter

Solution: Pass the size and use it

- And don't bother to specify the size of the array formal parameter
  - But note, the **SIZE IS NOT CHECKED**
  - Can enter the size as a constant or calculate it using `sizeof`
    - `Sizeof` returns the number of bytes used by a variable or type

```
1  //Array size parameter
2  #include <iostream>
3  using std::cout; using std::endl
4  int sumAll(int a[], int size){      // Pass the size
5     int ans = 0;
6     for (int i = 0; i < size;)
7        ans += a[i++];
8     return ans; };
9  };
10
11 int main(){
12    int arr1[6] = {11, 12, 13, 14, 15, 16};
13
14    cout << sumAll(arr1, 6) << endl;
15
16    cout << sumAll(arr1, sizeof(arr1) / sizeof(int)) << endl;
17 }
```

- Passing the size as a constant (ie 6) assumes that the size is correct
  - Calculating the size using `sizeof` is guaranteed to give a valid size
- See the section on Pointers for more information on arrays

9.5 Arrays are passed by reference

Arrays are automatically passed by reference. Changes in an array parameter are reflected in the called (like Ada `in out`).

```
1  void clearFirst(int a[]){
2     a[0] = 0;
3  };
4  int main(){
5     int arr[6] = {11, 12, 13, 14, 15, 16};
6
7     cout << arr[0] << endl;                  // 11
8
9     clearFirst(arr);
10
11    cout << arr[0] << endl;                  // 0
12 }
```

10 Records (ie structs)

Structs are similar to Ada records

```
1  //Structs
2  #include <iostream>
3
4  struct Pair  // Like an Ada record
5  {
6     int x;
7     int y;
8  };
9
10 int sum(Pair p){return p.x + p.y;};
11
12 int main(){
13
14    struct Pair r;   // C requires the word struct
15    r.x = 11;
16    r.y = 12;
17
18    Pair s;          // C++ does not require the word struct here
19    s.x = 21; s.y = 22;
20
21    Pair t{31, 32};  // Initializer
22
23    std::cout << sum(r) << sum(s) << sum(t) << std::endl;  //23 43 63
24
25    struct TwoPairs
26    {
27       Pair first;
28       Pair second;
29    } myPairs{{1,2},{3,4}};
30
31    std::cout << sum(myPairs.first)+sum(myPairs.second); //10
32
33 }
```

- 4: The type `struct Pair` is defined to be global so that function `sum` can be defined
  - If this type had been defined inside `main`, but then the function `sum` could not be defined
- 25: The definition of `struct TwoPairs` includes a definition and initialization of the variable `myPairs`.
- In C, the word "`struct`" is required in the variable declarations, as in the declaration of p
  - This restriction is relaxed in C++
  - A **typedef** can be used to define a different name in C (and C++)

11 typedef

C uses **typedef** to define alternate names for types.

```
1  //Structs and typedef
2  #include <iostream>
3  using std::cout; using std::endl;
4
5  struct Pair_Definition
6  {
7     int x;
8     int y;
9  };
10
11 typedef Pair_Definition Pair;
12
13 int sum(Pair p){return p.x + p.y;};
14
15 int main(){
16
17    Pair r;
18    r.x = 11;
19    r.y = 12;
20
21    Pair s;
22    s.x = 21; s.y = 22;
23
24    Pair t{31, 32};
25
26    cout << sum(r) << sum(s) << sum(t) << endl;  //23 43 63
27 }
```

The struct definition and the typedef can be combined

```
1  //Structs
2  #include <iostream>
3
4  typedef struct Pair_Definition
5  {
6     int x;
7     int y;
8  } Pair;
9
10 Pair p{1,2};
```

- Hint: In a typedef, the name of the new type goes where a variable would go if defining a variable of the type.

12 Pointers, Heap Variables, new, and -> notation

12.1 Pointer types (pointing to an int on the stack)

Example of a pointer (like Ada: `type intptr is access Integer; ip:  IntPtr;`)

```
1  //Structs
2  #include <iostream>
3  using std::cout; using std::endl;
4
5  int main(){
6     int * ip;      // Pointers to an int
7
8     int i = 55;
9     ip = &i;       // ip points to local i (allocated on stack)
10
11    cout << *ip;  // Dereference ip and jp.  Output is 55
12 }
```

- 6: Declare a pointer variable: Variable `ip` points to an integer
  - `*` means dereference
  - Intuition: When you dereference `ip` you get an integer
- 9: Assign a pointer value: `ip` points to `i`
  - `ip` points to a local variable (ie allocated on the stack)

12.2 Type checking

Some type checking is done: `ip` must point to an int, not a double:

```
1     int * ip;
2     int i = 55;
3     ip = &i;      // Compiles
4
5     double dd;
6     ip = &dd;     // Compile error
```

12.3 Pointing to variables on the heap

Pointer to an anonymous variable on the heap, allocated with new:

```
1    int * ip, jp;          // Two pointers
2
3    int i = 55;
4    ip = &i;               // ip points to a stack-allocated variable
5
6    jp = new int{66};      // jp points to a heap-allcated
7
8    cout << *ip << *jp;    // Dereference ip and jp. Output: 55 66
```

- 6: Dynamically allocate a value with new and initialize it to 66
- Comparison with Ada:
  - Ada has pointer types (eg `type PairPtr is access Pair`). C/C++ has no pointer types; it only has pointer variables and parameters.
  - In Ada, special syntax is required for a pointer to point to a stack (ie local) variable. In C/C++, any pointer can point to any variable

12.4 Pointing to a struct with ->

A pointer to a struct can use a -> to dereference and access a field

```
1  #include <iostream>
2  using std::cout; using std::endl;
3
4  int main(){
5     struct Pair{
6        int x; int y;
7     };
8
9     Pair aPair{11,22};
10
11    Pair * pp = &aPair // pp points to pair aPair
12
13    cout << p->x << p->y << endl;         // 11 22
14
15    cout << (*p).x << (*p).y << endl;     // 11 22
16
17    // Heap allocated Pair
18    Pair * hp = new Pair{77, 88};
19    cout << hp->x << hp->y << endl;        // 77 88
20 }
```

- 15, 21: Arrow notation: dereference pointer to struct and access field
- 17: Can also defererence as usual (ie (`*p.x`))
  - Use parentheses for precedence

12.5 Heap Allocated Arrays

```
1  //Arrays
2  #include <iostream>
3
4  int main(){
5     int size = 5;
6
7     //  Heap allocated
8     int* arr1 = new int[size];
9     int* arr2 = new int[size]{0};
10    int* arr3 = new int[size]{11, 12, 13, 14, 15};
11
12    int sum = 0;
13    for (int i = 0; i < size; i++)
14       sum += arr3[i];
15
16    std::cout << sum << std::endl; // 65
17 }
```

- The arrays are allocated on the heap
  - 10: `int* arr3` means that `arr3` is a pointer to the first element of the array
  - Access an element of `arr3` using regular array notation (ie `arr3[i]`)
  - See below for more information on accessing arrays via pointers, on the heap or on the stack
  - 10: Note: Initializer for `arr3`
  - Ada comparison: Ada distinguishes between using pointers to access the stack and the heap (ie pointers that access the stack must be declared as as in `type intptr is access all Integer;`, and local variables must be declared as having pointers to them, as in `i: aliased Integer;`)

13 Pointers and Arrays, Array Parameters (revisited), Malloc

13.1 Accessing arrays with pointers

Arrays can be accessed using pointers

```
1  //Arrays
2  #include <iostream>
3
4  int main(){
5     int size = 3;
6
7     int stackbased [size] = {11, 12, 13};
8
9     //  Heap allocated
10    int* heapbased = new int[size]{11, 12, 13};
11
12    int sum = 0;
13    int* ptr = heapbased;
14    for (int i = 0; i < size; i++)
15       sum += *ptr++;   //  ++ increments by 4!!!!!
16
17    std::cout << sum << std::endl;  // 36
18
19    sum = 0;
20    ptr = stackbased;
21    for (int i = 0; i < size; i++)
22       sum += *ptr++;
23
24    std::cout << sum << std::endl;  // 36
25 }
```

- 10: Can explicitly declare a pointer (ie `ptr`) that points to the first array element and access the array elements using pointer dereferencing.
- 15: Notice that `ptr++` increments by 4, the size of one int on the array!!

13.2 Pointers, arrays, parameters

When an array is passed as a parameter, the formal parameter gets the address of the first element. The actual array can be accessed using either normal array notation (ie `a[i]`) or with pointer dereferencing (ie `*a++`). (Arrays are passed by reference, as noted above.)

```
1  //Array parameters accessed via pointers
2  #include <iostream>
3  using std::cout; using std::endl
4  int sumAll(int a[], int size){
5     int ans = 0;
6     for (int i = 0; i < size; i++)  // increment i here (vs below)
7        ans += *a++;
8        // a is a pointer to first element
9        //   so can access the array using pointer dereferencing
10       // a++ advances to next array element
11
12       //ans += a[i++];  //  Regular access, not required
13    return ans; };
14 };
15
16 int main(){
17    int arr1[6] = {11, 12, 13, 14, 15, 16};
18
19    cout << sumAll(arr1, 6) << endl;                          // 81 = sum(11 .. 16
20
21    cout << sumAll(arr1, sizeof(arr1) / sizeof(int)) << endl;  // 81
22 }
```

13.3 malloc and sizeof

From C, C++ gets the function `malloc`, meaning memory allocation, to allocate on the heap, similar to Ada `new`. `Malloc` allocates memory and returns a pointer to that memory. It is used in conjunction with `sizeof`, which gives the size in bits of its parameter.

```
1  int* intptr = (int*)malloc(sizeof(int));
2
3  *intptr = 99;      // assign 99 to the location at which intptr points
4  (*intptr)++;       // dereference then increment.
5                     //   * has lower precedence so use ()
6  cout << *intptr;   // 100
```

- `intptr` is a pointer to an integer.
- `sizeof(int)` gives the number of bytes of an int (probably 4)
- `malloc` allocates the number of bytes used by an int (ie 4)
- `(int*)` - casts the pointer returned by `malloc` into the needed type
  - Before the cast it pointer is of type `void*`

This example allocates space for a string of size 4, plus one for the zero termination:

```
1 char * mystr = (char *) malloc (5);
2 mystr = "abcd";
```

14 Classes

14.1 Classes and Structs

At its simplest, a class is essentially the same as a record, except that the fields are private by default.

- The following example implements the pair example using a class.

```
1  //Classes are like structs
2  #include <iostream>
3
4  class Pair
5  {
6     public:     // Default is private
7        int x;
8        int y;
9  };
10
11 int main(){
12
13    Pair r;
14    r.x = 11;
15    r.y = 12;
16
17    Pair s{31, 32};
18
19    std::cout << r.x << s.y << std::endl;  //234363
20 }
```

- The values (ie `x` and `y`) inside the class `Pair` are called the fields of the class.
- The values allocated for `Pairs` `r` and `s` are exactly the same as they would be if type `Pair` were defined as a `struct`.

14.2 Classes with function members

- Like Java, classes can contain functions as well as fields
- Typically, signature declarations are inside the class and the associated function definitions are outside the class.
  - Short function definitions are sometimes defined inside the class.

```
1  //Inheritance and overriding
2  #include <string>
3  #include <iostream>
4  using std::string;
5  using std::to_string;
6  using std::cout;
7  using std::endl;
8  class pair{
```

```
9     public:
10       // Constructor
11       pair(int x=0, int y=0) // default values{
12          _x = x;
13          _y = y;
14       }
15
16       int sum(){return _x + _y;};  // Simple definition
17
18       void reverse();              // Declarations only
19       virtual string toString();   // Polymorphism possible
20    private:
21       int _x;
22       int _y;
23 };
24
25 //  pair:: denotes that this is part of class pair
26 void pair::reverse(){              // Implementation
27    const int t = _x;
28    _x = _y;
29    _y = t;
30 }
31
32 //  pair:: denotes that this is part of class pair
33 string pair::toString(){           // Implementation
34    string answer = "(" + to_string(_x) + ",␣" + to_string(_y) + ")";
35    return answer;
36 }
37
38 // Inherit from  pair, making pair's elements public
39 class coloredPair : public pair{
40    public:
41       // Constructor signature
42       coloredPair(int x, int y, int color); // no default values
43
44       // Override pair::toString
45       virtual string toString();
46
47    private:
48       int _color;
49 };
50
51 //  Constructor implementation // No default values
52 coloredPair::coloredPair(int x, int y, int color)
53          : pair(x, y)   // Call base class 2 param constructor
54 {
55    _color = color;
56 }
57
58 //  overriding toString implementation
```

```
59 string coloredPair::toString(){
60     string answer = "[" + pair::toString()
61         + "; " + to_string(_color) + "]";
62     return answer;
63 }
64
65 int main(){
66     coloredPair c{61, 62, 63};
67     cout << c.toString() << c.sum() << endl;
68     // [(61, 62); 63]123
69
70   pair[2]
71 }
```

14.3 Polymorphism: Ada, C++, Java

Polymorphism (aka dynamic dispatch) occurs when the specific version of a method is chosen at runtime, based on the actual type of object. So polymorphism occurs when, for example, `toString()` is called on each element of an array of pairs (each of which can be either a `Pair` or a `ColoredPair`). .

```
// Declare and initialize an array of pointers to pairs
pair * a[2] = { new Pair(11, 12), new ColoredPair(13, 14, 15)};

// Polymorphism: Tostring based on kind of pair
cout  << a[0]->toString() << endl;  // (11, 12)
cout  << a[1]->toString() << endl;  // [(13, 14); 15]
```

However, polymorphism does not happen on all method calls, and the rules for when it occurs are language specific:

- Java: Polymorphism occurs for every overridden method
- C++: Polymorphism occurs only on methods that are declared as `virtual` (and when defined with pointers)
- Ada: Polymorphism can occur with any method, but it only occurs if the invoking variable is class-wide (rather than being of a specific type)

15 Union Types

Ada variant records are similar to union types. But union types are not typesafe.

16 Templates

Similar to Java's **parameterized types**. and have a similar function to Ada generics

17 Some More Syntax, Common Errors and Reliable Programming

17.1 Some more syntax

- = is an operator that returns a value (ie the value on the expression on the right of the assignment
  - `x = y = 3; // assigns 3 to both x and y`
  - `while (c = getc()){putc(c);} // This program echo prints its input.`
    - getc() and putc() are from the C library stdio
- `x - 3;` // Yes, this compiles
  - C and C++ allow expression results to be ignored
- C strings are zero-terminated (ie a zero byte marks the end of the string), causing LOTS of problems
  - C++ has a string library
- Help: Lint

17.2 Common errors

- Segmentation fault: typically trying to dereference an address that is outside the program's address space
- Bus error: typically trying to access an address that isn't divisible by 4
- Memory clobber: Can overwrite memory if accessing an array out of bounds
- Link error on compilation step:
  - Undefined reference: occurs when a header exists, allowing compilation, but the definition does not exist at link time

17.3 Reliable programming

- MISRA C is a standard for reliable C programming
- C and C++ have lots of Undefined Behavior, making reliable programming difficult
  - Ada SPARK has NO undefined behavior!

18 Summary Table

Ada	C++
package specification (.ads file)	header file (.h file)
mymain.adb, mypkg.ads, mypkg.adb	mymain.cpp, mypkg.h, mypkg.cpp
gnatmake mymain	gcc mymain.cpp, mypkg.cpp
executable: mymain [.exe]	executable: a.[out | exe]
if a /= b then x := y elsif c = d then x := 1; y := 1 end if;	if (a != b) x=y else if (c == d){x=1;y=1}
for i in 1 .. 10 loop ... end loop;	for (int i=1; i<=10;i++){...}
Integer, Float (32, 32 bits) [4 bytes]	
Long_Integer, Long_Long_Integer (32, 64) [4, 8]	int, long int, long long int
Long_Float, Long_Long_Float (64, 128) [8, 16]	float, double, long double
	Above types can be designated signed or unsigne
type Mod32 is mod 32; m: Mod32;	short
d: Float := 1; – Compile error	double d = 1; // implicit cast
subtypes: Natural, Positive	byte, short ??
Boolean	bool
No implicit conversion (except between subtypes)	Implicit conversions (eg between bool and int)
c: Constant Integer := 3;	const int c = c;
type foo is record a,b: integer; end record; r: foo;	struct foo{int a; int b;}; foo r;
type arr is array(1 .. 10) of Integer;	int[] arr;
procedure foo(a: integer) is ... begin ... end foo;	void foo(int i){...}
function foo(a: integer) return integer is..begin..end foo;	int foo(int i){...}
procedure foo(a: in out Integer) is ...	void foo(int &i){...}
with ada.text_io; with ada.integer_text_io;	#include <iostream>
ada.text_io.put("Hi");	std::cout("Hi");
use ada.text_io;	using std::cout