CMPUT 350

Advanced Game Programming: C++

Instructor: Michael Buro

These are notes for the C++ part of CMPUT 350 as it was taught at the University of Alberta in the Fall term of 2020. The presented material has been drawn in part from freely available sources, such as Wikipedia and C/C++ tutorial webpages, and notes I collected during years of C/C++ programming and teaching.

These notes go beyond the slide shows that are usually presented in course lectures these days in an attempt to replace costly text books.

I appreciate suggestions for improvements.

– Michael Buro, Edmonton, September 2020

C++ Course Contents

Part 1: From C to C++

Introduction, C vs. C++ overview, reference types, const, default arguments, dynamic memory allocation (`new`/`delete`)

Part 2: C++ Classes

C structs vs. C++ classes, separating interface and implementation, constructors, desctructors, copy constructor, assignment operator

Part 3: Object Oriented Programming in C++

Class inheritance, assignments across class hierarchy, reusing base class operators, constructors/destructors and inheritance, virtual functions

Part 4: More on C++ Classes

Type casts, static data and functions, operator overloading

Part 5: Generic Programming in C++

Template functions, class templates, template specialization, template applications

Part 6: Standard Template Library (STL)

Sequence containers, associative containers, STL algorithms

Part 7: C++ Odds and Ends

Exceptions, RAII, Smart Pointers, major C++11 / 14 / 17 additions, Review

Part 1: From C to C++

Contents [DOCUMENT NOT FINALIZED YET]

Introduction

Why C++? you may ask.

As the name suggests, C++ strives to be a better C.

C pros:

- Compilers are FAST; code is FAST; often only little slower than hand-written assembly language code
- Lingua Franca of computing — C is ubiquitous
- Portable. C compilers are available on all systems
- Compilers/interpreters for new languages are often written in C

Some C issues C++ rectifies:

- Struct initialization and freeing resources is error prone
  (must not forget calling functions)
- Weak support for generic programming
  (essentially macros)
- Object oriented programming not well supported

C vs. C++

We assume that you have basic knowledge of C (remember CMPUT 201 or 274/5?)

Feeling rusty? Look at CMPUT 201 notes and CRefresher.pdf (on webpage)

C can be considered a subset of C++, with only a few exceptions. E.g.

```
void *ptr;
int *i = ptr; // valid C but not C++
        // can't assign generic pointers (void*)
        // to other pointer variables
```

C++ additions:

- reference types, const correctness,
- default parameter values,
- classes, inheritance,
- operator overloading,
- templates, exceptions, name spaces,
- and more

[The latest standard (C++17) supports lambda expres-

sions, auto, initializer lists, and much more]

These make it much easier to manage large projects

Code can be made safer and more readable without sacrificing speed. C++ implements the "zero-overhead" principle: you don't pay for features you don't use

```
// this is a C program
#include <stdio.h>

int main()
{
  printf("hello world\n");
  return 0;
}
```

```
// this is a C++ program
#include <iostream>

int main()
{
  std::cout << "hello world" << std::endl;
  return 0;
}
```

Basic Building Blocks

```
// this is a comment

#include <iostream> // preprocessor command: include file
// mostly used for functions, macros, class declarations

int foo(int x) // function definition: return type
               // and parameters
{              // block
  return x+1;  // return expression value
}

int main()     // this is where all C/C++ programs start
{
  int i = 0;           // variable declaration + init.
  while (i < 10) {     // loop
    i = i+1;           // expression + assignment
    std::cout << foo(i) << " ";  // operators + call
  }
  if (i >= 10) {       // condition
    i = 1;
  } else {
    i = 0;
  }
  return i;            // return result, exit function
}
```

It is important to comment your code – for others and yourself!

```
/* this is an old-style
   multi-
   line
   comment (C, also C++)
*/

// this is a single line comment (C++)
```

Multi-line comments cannot be nested; not allowed:
`/* /* */ */`

Where to put comments?

- mostly in header files where functions and structs/-classes are defined (this is where the user looks, who often is not interested in implementation)
- at the beginning of files describing their purpose
- on top of function definitions discussing parameters, function effects, and return values
- on top of struct/class definitions describing their purpose
- in front of non-trivial parts, meaning anything you

wouldn't instantly understand when looking at the code a month later

There is no need to write novels or to comment each program statement!

Introduction to C++ Input/Output

```
#include <iostream>  // imports definitions of
                     // input/output stream classes
using namespace std; // instructs compiler to look in
                     // namespace std, in which cout
                     // and endl are defined.
                     // Saves typing: E.g. cout rather
                     // than std::cout
int main()
{
  int n;
  cout << "n=?\n";                  // output string
  cin >> n;                         // input number
  cout << "2*n=" << (2*n) << endl;  // output string and
                                    // expression result
  return 0;
}
```

Input via input-stream cin ("standard input")

```
cin >> var1 >> var2 >> ...  >> varn;
```

Output via output-stream cout ("standard output")

```
cout << exp1 << exp2 << ...  << expn;
```

`cin`/`cout` defined in standard C++ header file `<iostream>`

[similar to stdin/stdout in C (file descriptors 0,1)]

Standard Error Stream

Another predefined output stream: `cerr`

Used for error messages

Same output operator: <<

By default, output is also sent to the console, but it is not redirected when using > or |

```
cerr << "division by zero" << endl; exit(10);
```

[similar to stderr in C, file descriptor 2]

Can be redirected, too. E.g.

```
command > coutfile 2> cerrfile
```

sends output to cout to coutfile and output to cerr to cerrfile

```
command 2>&1 file
```

sends all output to cout and cerr to file

[see bash manual for more on redirection]

Visit `http://www.cplusplus.com/ref/iostream` to get more information on iostreams

C/C++ Number Types

C++ uses the same fundamental number types as C: char, short, int, float, double

C++ adds type bool which contains values true, false

IMPORTANT: make sure variables have the right type to avoid underflows and overflows

In C/C++, integer expressions are NOT checked for overflows/underflows!

```
unsigned char foo = 255; // unsigned 8-bit value
unsigned char bar = foo+1;
// value of bar is 0 because of 2s-complement
// number representation:
// 255 (base 10) = 11111111 (base 2)
//                +00000001
//            =(1) 00000000 = 0 (base 10)
```

Floating-point overflows/underflows are indicated by special values (+Inf, -Inf, NaN)

But program continues anyway (even after say c = 1.0/0.0)

Reference Types

C passes parameters by value (except for arrays), C++ supports parameter references directly

C++ references are **aliases**. They can be emulated in C by passing pointers to variables

Example 1: Using references

```
// & prefix indicates reference
int A[] = { 1, 2, 3, 4 };
int &x = A[2]; // x is a reference (or alias) of A[2]
++x;           // operations on x now affect A[2] (now 4)
if (x > 10) ...   // equivalent to if (A[2] > 10) ...
```

Example 2: Call-by-reference

```
void increment(int &x)
{
  ++x;
}

int y = 5;
increment(y);  // that worked: y now 6
```

Here, a reference to variable y is passed to a function (internally represented as a pointer to y). In this case reference x is an alias for variable y

Statements in the function body that act on the parameter change the variable whose reference has been passed. So, by using references, a function can have side effects (i.e., change the calling environment)

Compare this to

```
void nop(int x) // call by value
{
  ++x;
}
```

Here, local variable x is incremented, but then discarded when the function is exited. Function nop doesn't change anything in the caller environment

Using reference parameters, we can return more than one function value

By convention, input parameters are listed first, followed by output (reference) parameters

```
// function "returns" 3 values
int foo(int in1, int in2, int &out1, int &out2)
{
  out1 = out2 = 3;    // 2
  return 4;           // one more
}
```

Only variables can be passed on to reference parameters, because an address is required

Example: Swap Function

```
void swap(int &x, int &y)
{
  // triangle exchange
  int temp = x; x = y; y = temp;
}

a = 1; b = 2; // before: a = 1, b = 2
swap(a, b);   // after:  a = 2, b = 1

swap(1, 2);   // error
```

Equivalent C code:

```
void swap(int *px, int *py)
{
  // triangle exchange
  int temp = *px; *px = *py; *py = temp;
}

a = 1; b = 2; // before: a = 1, b = 2
swap(&a, &b); // after:  a = 2, b = 1
```

Sometimes it is useful to be able to pass constants or results of function calls as arguments to functions with reference parameters. E.g.

```
int foo();
void bar(int &x);
...
bar(2); bar(foo());
```

However, both forms are illegal in C++, because constants and return values are nameless temporary objects (so-called "right-hand-side" or rvalues). If function bar has a side effect on parameter x, this effect wouldn't be visible when applied to temporary objects because they are destroyed before the next expression is evaluated

Using `void bar(const int &x)` solves this problem

Passing Large Read-Only Objects

```
void do_something(T big) { ... }
...
T x;
do_something(x); // slow! x is copied
```

Passing large read-only objects by value is wasteful: they are copied into local variables and not changed ...

Use const reference instead:

```
void do_something(const T &big) { ... }
...
T x;
do_something(x); // equivalent but much faster!
```

Address is passed to function, no copy overhead

const ensures that the function body does not change the object. If it does, the compiler will issue an error message

Call-by-Value vs. Call-by-Reference

Call-by-Value

+ Callee detached from caller, no direct side effects
– Data is copied to a local variable. Can be time consuming

Call-by-Reference

– Possible side effects, need to look at function declaration to see whether call-by-reference is used
  E.g. foo(x) doesn't tell you whether foo's parameter is `int` or `int&`
+ Only reference is copied. Fast - internally it's just a pointer

Added bonus: const qualifier protects read-only parameters

Default Arguments

```
void print(int value, int base = 10);

print(31);    print(31,10);
print(31,16); print(31,2);

->  31 31 1f 11111
```

Arguments can have default values

All default arguments must be in the rightmost positions. Omitting arguments begins with the rightmost one. E.g.

```
void foo(int a, int b=2, int c=3, int d=4) {...}

foo();              is illegal
foo(x);             calls   foo(x, 2, 3, 4);
foo(x, y);          calls   foo(x, y, 3, 4);
foo(x, y, z);       calls   foo(x, y, z, 4);
foo(4, 3, 2, 1);    calls   foo(4, 3, 2, 1);

// illegal:
void bar(int a=1, int b, int c=3, int d);
// why? bar(x,y,z) would be ambiguous - is c or
// a assigned a default value?
```

Dynamic Memory Allocation

Local variables, function parameters, and function call return addresses are located on the runtime stack (last-in first-out (LIFO) data structure)

Dynamic memory that is more permanent and can outlast function calls is allocated from a different part of memory called heap

Operator `new` dynamically allocates memory on the heap

Operator `delete` is used to release it when no longer needed – can be done later, even in a different function

As always with C or C++, YOU are in control because the compiler cannot know when memory is no longer needed and can be deleted

C/C++ does not have a garbage collector (yet)

Operator new

```
int *p = new int;          // allocates space
                           // for one int
                           // p now points to it

// if program gets here, allocation succeeded

*p = 0;                    // use allocated memory
```

There is no initialization for basic C — plain old data ("POD") – types, unlike Java or Python!

YOUR CODE NEEDS TO INITIALIZE POD EXPLICITELY. OTHERWISE CONTENT IS UNDEFINED, IN WHICH CASE YOU CAN EXPECT RANDOM PROGRAM BEHAVIOUR!

Calling `new` with class type calls class constructor (later)

No need to check return value against 0 — if no memory is available an exception is thrown (later)

Good practice: don't use `malloc` and `free` in C++ programs. `new`/`delete` are as fast and easier to use!

Operator delete

```
int *p = new int; // allocate one int on heap

// do something with integer p points to (*p)
...

// free memory when integer *p is no longer used
delete p;
```

delete frees the memory its parameter points to

If p == 0, delete p does nothing. So you don't have to check p before calling delete

Before returning the memory back to the operating system, the class destructor for non-POD types is called (later)

Good practice for debugging your code: set pointer to nullptr after delete to prevent further access of this address through this pointer

```
delete p;
p = nullptr;  // using p later will cause
              // segmentation fault
```

Also: make sure each heap object has exactly one owner

(i.e., object with a pointer pointing to the object) that is responsible for its deletion

In C and C++03, 0 (zero) is a special pointer value that can be assigned to any pointer variable regardless of type

0 is not the address of any process memory. It can therefore indicate errors when used as function return value, or special conditions such as "this linked list node has no successor"

0 is also an integer constant, which sometimes leads to ambiguities (is it a pointer or is it an integer?)

Since C++11, using C's NULL or 0 as pointer value is discouraged. Use nullptr instead

Dynamic Arrays

```
const int N = 100;        // C++ way of declaring constants
float *p = new float[N]; // allocate N consecutive
                          // floating point numbers
...
for (int i=0; i < N; ++i) {
  p[i] = 0.0;
}
...

// free array when no longer used
delete [] p;
```

`new[]` allocates an array of elements of the given type on the heap

POD variables are not initialized, but for non-POD variables (i.e., classes) the constructor is called for each array element. We'll look at constructors shortly

When no longer used, free arrays using `delete[]`

Before memory is released, `delete` calls destructor for each array element if it is non-POD

`new`/`delete` Match

`new`/`delete` come in pairs:

For every `new` there should be at least one `delete` in your program to avoid memory leaks

More specifically:

- For every `new` at least one corresponding `delete`

- For every `new[]` at least one corresponding `delete[]`

If mixed, the computation result is undefined

Such bugs are hard to track. Tools like valgrind and setting pointers to `nullptr` after `delete` can help