Sizeof and Vtables; Modern C++ Topics

ITP 435

Week 3, Lecture 1

Sizeof and Vtables

sizeof Operator

Tells you the number of bytes a particular data type (or variable) takes up:

```
// Returns 4
sizeof(int)

char a;
// Returns 1
sizeof(a)

// Returns 1
sizeof(bool)
```

sizeof Question #1

```
int* ptr;
ptr = new int[10];
```

Q: What is the `sizeof(ptr)`??

A: On a 32-bit system, 4. On a 64-bit system, 8 (we'll stick with 64-bit).

sizeof Question #2

```
int array[10];
```

Q: What is the `sizeof(array)`??

A: 40, because each `int` is 4 bytes in size.

sizeof Question #3

```
int array[10];
int* ptr = array;
```

Q: What is the `sizeof(ptr)`??

A: It’s a pointer, so you’ll get either 4 or 8 again!

sizeof Question #4

```
class Test1
{
      char c;
      int i;
};
```

Q: What is the `sizeof(Test1) ??`

A: 8, because of padding.

Class Padding

- By default, Visual Studio (and most compilers) guarantees alignment to be equal to the size of the largest basic type in the struct/class
- ***x-byte aligned*** means the variable's memory address is guaranteed to be divisible by x
- So our previous example, we have a 4 byte variable (`int i`) that must be 4-byte aligned. Thus, we need padding:

Test1	
c	3 BYTES
i	

More Padding

What happens with this?

```
class Test2
{
      char a;
      int i;
      char b;
};
```

Test2	
a	3 BYTES
i	
b	3 BYTES

`sizeof(Test2) == 12`

The compiler won’t rearrange variables!!

Even More Padding

Solution: Always declare your variables from largest to smallest:

```
class Test3
{
      int i;
      char a;
      char b;
};
```

Test3		
i		
a	b	2 bytes

sizeof Question #5

Q: What's the sizeof this class?

```
class VirtualClass
{
	int i;
	virtual void F();
};
```

A: 16 (for a 64-bit program). Because it has a virtual function, the size of the class is the pointer to a virtual function table + the int+ padding.

sizeof Question #5

Q: What’s the sizeof this class?

```
class VirtualClass
{
      int i;
      virtual void F();
};
```

VirtualClass	
vtable*	
i	4 bytes

A: 16 (for a 64-bit program). Because it has a virtual function, the size of the class is the pointer to a virtual function table + the int+ padding.

Cost of Virtual Functions

Memory cost:

- Need to add 1 pointer to the start of a class' data once it has virtual functions.
- This points to the "virtual function table" or vtable.

Performance cost:

- At run-time, when a virtual function is called virtually, the vtable pointer is dereferenced, which then finds the correct function to call from the vtable.

Padding and Aggregate Types

```
struct A {
   double d;
   char s;
};
struct X {
   virtual void a();
   virtual void b();
   void c();
   virtual ~X();
   A myA;
};
```

X	
vtable* (8)	
myA.d (8)	
myA.c (1)	7 bytes padding

- Even though the sizeof(A) is 16, the "row size" is still 8 because we only care about the largest basic type inside the aggregate or in the current class

Virtual Function Tables

- Simple example:

```
class A
{
public:
   virtual void x();
   virtual void y();
   virtual ~A();
};

class B : public A
{
public:
   virtual void x();
   virtual void z();
   virtual ~B();
};
```

vtable for A (24 bytes if 64-bit)	
Index	**Function Pointer**
0	&A::x()
1	&A::y()
2	&A::~A()

vtable for B (32 bytes if 64-bit)	
Index	**Function Pointer**
0	&B::x()
1	&A::y()
2	&B::~B()
3	&B::z()

A couple more sizeof/vtable examples…

```
class X
{
   int i;
public:
   virtual void a();
   void b();
   virtual void c();
   virtual ~X();
};
```

```
sizeof(X) == 16
```

X	
vtable*	
i	padding

vtable for X (24 bytes if 64-bit)	
Index	**Function Pointer**
0	&X::a()
1	&X::c()
2	&X::~X()

A couple more sizeof/vtable examples…

```
class Y : public X
{
   char p;
public:
   virtual void c() override;
};
```

```
sizeof(Y) == 24
```

Y	
vtable*	
i	padding
p	padding

vtable for Y (24 bytes if 64-bit)	
Index	**Function Pointer**
0	&X::a()
1	&Y::c()
2	&X::~X()

In-class activity

```
class A
{
public:
   virtual void f();
   virtual void g();
   void h();
private:
   char c;
   double d;
};
```

```
class B : public A
{
public:
   void f() override;
   void h();
   virtual void i();
private:
   int i;
};
```

sizeof Question #6

Q: What's the sizeof this class?

```
class Empty
{
};
```

A: 1. Classes cannot take up 0 memory.

Union

- A ***union*** is a struct that can hold only one of its members at any time*
- Example:

```
union MyUnion
{
    // Only can hold one of these at a time
    int n;
    float f;
    char c;
};
```

*Technically, the union just takes the same chunk of memory and can treat it as different types as desired

Example

```
MyUnion test;
test.n = 50;
```

MyUnion
50

Example

```
MyUnion test;
test.n = 50;

test.f = 2.5f;
```

MyUnion
2.5f

Example

```
MyUnion test;
test.n = 50;

test.f = 2.5f;

test.c = 'a';
```

MyUnion
'a'

Sizeof Union

- The sizeof a union is equal to the size of the largest member of the union:

```
union MyUnion
{
   int n;    // 4 bytes
   float f; // 4 bytes
   char c;  // 1 byte
};
```

- sizeof(myUnion)????
- 4

Why use a union?

There's a couple of cases where it might be useful, maybe

1. Addressing the same chunk of memory in multiple ways:

```
union PointUnion
{
    struct { float x, y, z; } p;
    float a[3];
};

// Later...
PointUnion pt;
pt.p.x = 5.0f;
std::cout << pt.a[0]; // 5.0f
```

Why use a union? Cont'd

2. Can use to pack in data that's mutually exclusive:

```
struct SomeFileType
{
   enum SubType { v1, v2 };
   SubType type;
   union SomeUnion
   {
      // Could be one of these, based on sub type
      struct { /*  */} s1;
      struct { /*  */ } s2;
   };
};
```

Why use a union? Cont’d

- In practice, sort of an edge case
- More of a C thing than C++
- YMMV

Modern C++ Topics

constexpr

- C++11 feature to allow *compile-time* computation
- Example:

```
constexpr int max(int a, int b)
{
     return (a > b ? a : b);
}
```

- Then if we have code like this:

```
constexpr int a = max(5, 6);
```

- The *compiler* will replace it with:

```
int a = 6;
```

constexpr

- Better example:

```
constexpr int factorial(int x)
{
     if (x > 0)
     {
          return x * factorial(x – 1);
     }
     else
     {
          return 1;
     }
}
```

An Annoyance

- This won't compile:

```
struct Test
{
    static const int CONST_INT = 0;
    static const float CONST_FLOAT = 0.0f;
};
```

- Error: "A member of type const float cannot have an in-class initializer"

But this does!

```
struct Test
{
    static constexpr int CONST_INT = 0;
    static constexpr float CONST_FLOAT = 0.0f;
};
```

Effective Modern C++ says

- “Item 15: Use constexpr whenever possible”

- I think this is a decent idea

Code Samples

- Some samples here:
- https://github.com/chalonverse/CPPBeyondSamples

Filesystem
(Ex02 in repo)

Filesystem

- Purpose:

"implementations of an interface that computer programs written in the C++ programming language may use to perform operations on file systems and their components, such as paths, regular files, and directories."

- Gives us a cross-platform way for common file system operations like finding if a file exists, getting the size of files, iterating over files in directories, and a lot more!

Checking if Files exist

```cpp
// Shortcut so we don't have to type out std::filesystem every time
namespace fs = std::filesystem;

// Get the size of a file (throws an exception if it doesn't exist)
try
{
   auto fileSize = fs::file_size("CMakeLists.txt");
   std::cout << "Size = " << fileSize << "\n";
}
catch (fs::filesystem_error& e)
{
   std::cout << "Error from filesystem: " << e.what() << "\n";
}
```

No exceptions?

- If you don't want to use exceptions, most versions of filesystem functions have an alternative version that takes in an error_code:

```
std::error_code ec;
auto fileSize = fs::file_size("asdf", ec);

// A default error code means it's ok
if (ec == std::error_code{}) {
    std::cout << "Size = " << fileSize << "\n";
} else {
    std::cout << "Error: " << ec.message() << "\n";
}
```

Path objects

- You can construct a path object to a single file, and extract information from it:

```
fs::path pathTest("CMakeLists.txt");

// Does this file exist?
if (fs::exists(pathTest))
{
    std::cout << "File exists\n";
}
else
{
    std::cout << "File does not exist\n";
}
```

Iterating over all files in a directory

```
for (const auto& p : fs::directory_iterator(".")) {
    std::cout << p.path() << "\n";
}
```

Sample output:

```
".\\.git"
".\\.gitignore"
".\\.travis.yml"
".\\.vs"
".\\build"
".\\CMakeLists.txt"
".\\CMakeSettings.json"
".\\Ex01"
".\\Ex02"
".\\Ex03"
".\\Ex04"
".\\Ex05"
".\\Ex06"
".\\Ex07"
".\\README.md"
```

Optional, Variant, Any
(Ex03 in repo)

std::optional

- An optional is a wrapper that can ***optionally*** contain a value
- You can use this for cases where you're not sure whether or not something would happen, for example:

```
// This function will try to convert a string to an integer and either --
// Return the integer if successful
// Return an unset optional if not successful
std::optional<int> TryGetInt(const std::string& s) {
   try {
      int result = std::stoi(s);
      return std::optional<int>(result);
   } catch (std::exception&) {
      // stoi failed so return an unset optional
      return std::optional<int>();
   }
}
```

std::optional

- You can pass around optionals to other functions, and the bool conversion operator will be true if it’s set, false otherwise:

```
void CoutOptional(const std::optional<int>& o) {
   if (o) {
      std::cout << o.value() << "\n";
   } else {
     std::cout << "Optional was unset!\n";
   }
}
```

std::optional

- This code:

```
auto result1 = TryGetInt("10");
CoutOptional(result1);
auto result2 = TryGetInt("xyzw");
CoutOptional(result2);
```

- Would output:

```
10
Optional was unset!
```

std::variant

- This is C++'s answer to a union

- You set all the possible types as template arguments to the variant when you create it:

```cpp
// This variant can either contain an int or a string
std::variant<int, std::string> var;
```

std::variant

- You can set a variant just like a normal variable of any of the types
- To get the variant, use the get template function:

```
// Set it to an int
var = 20;
// We can use get<type>(variant) to extract the type
std::cout << "As int: " << std::get<int>(var) << "\n";

// Set it to a string
var = "Hello";
std::cout << "As string: " << std::get<std::string>(var) << "\n";
```

std::variant

- If you access the variant as a type it's not currently set to, you get an exception
- This means variants, unlike unions, are type-safe

```
// Set it to a string
var = "Hello";
std::cout << "As string: " << std::get<std::string>(var) << "\n";

// If we try to access it as a string we have an exception
try {
   int result = std::get<int>(var);
} catch (std::bad_variant_access&) {
   std::cout << "Error, variant wasn't set to an int!\n";
}
```

std::any

- Like variant, but not restricted to a specific type
- Not as efficient as using variant

```
// Initialize an any to any type
auto any1 = std::any(50);
// To extract the value we have to use any_cast
std::cout << "As int: " << std::any_cast<int>(any1) << '\n';
```

std::any

```
// If it doesn't contain the type, any_cast throws an exception
try
{
   std::cout << "As string: " <<
               std::any_cast<std::string>(any1) << '\n';
}
catch (std::bad_any_cast&)
{
   std::cout << "Any cast failed :(" << std::endl;
}
```

Feature Testing (C++20 and later)

Testing for features

- Need to include the <version> header
- Allows you to test, with the preprocessor, if a feature is supported, like:

```
#if defined(__cpp_lib_filesystem)
// This code is compiled if the platform
// support std::filesystem
std::filesystem::path myPath("hello");
#endif
```

Better way of checking for parallel algos

```
#if !defined(__cpp_lib_parallel_algorithm)
#if defined(__APPLE__)
// On Apple platforms, we can use this if we have to
#define PSTLD_HEADER_ONLY
#define PSTLD_HACK_INTO_STD
#include "pstld.h"
#else
#error "Parallel algos are required"
#endif // defined(__APPLE__)
#else
#include <algorithm>
#include <execution>
#endif // !defined(__cpp_lib_parallel_algorithm)
```