CSE 374 Programming concepts and tools

Winter 2024 Instructor: Alex McKinney

Review: `new/delete`

To allocate on the heap using C++, you use the `new` keyword

- You can use new to allocate an object (*e.g.* `new Point`)
- You can use new to allocate a primitive type (*e.g.* `new int`)
- When allocating you can specify a constructor or initial value
  - (e.g. `new Point(1, 2)`) or (e.g. `new int(333)`)
- If no initialization specified, it will use default constructor for objects, garbage for primitives (integer, float, character, boolean, double)
  - You don't need to check that `new` returns `nullptr`

To deallocate a heap-allocated object or primitive, use the `delete` keyword instead of **`free`**`()` from `stdlib.h`

- Don't mix and match!

Review: Dynamically Allocated Arrays

To dynamically allocate an array:

- Default initialize: `type* name = new type[size];`

To dynamically deallocate an array:

- Use `delete[] name;`
- It is an *incorrect* to use "`delete name;`" on an array
  - The compiler probably won't catch this, though (!) because it can't always tell if `name*` was allocated with `new type[size];` or `new type;`
    - Especially inside a function where a pointer parameter could point to a single item or an array and there's no way to tell which!
  - Result of wrong `delete` is undefined behavior

Template

Suppose that...

- You want to write a function to compare two `ints`
- You want to write a function to compare two `strings`
  - Function overloading!

```
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const int& value1, const int& value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const string& value1, const string& value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}
```

Hm...

The two implementations of `compare` are nearly identical!

- What if we wanted a version of `compare` for *every* comparable type?
- We could write (many) more functions, but that's obviously wasteful and redundant
  - Too much repeated code!

What we'd prefer to do is write "*generic code*"

- Code that is type-independent
- Code that is compile-type polymorphic across types

C++ Parametric Polymorphism

C++ has the notion of templates (often referred to as *generics* elsewhere)

- A function or class that accepts a **type** as a parameter
  - You define the function or class once in a type-agnostic way
  - When you invoke the function or instantiate the class, you specify (one or more) types or values as arguments to it

- At **compile-time**, the compiler will generate the “specialized” code from your template using the types you provided
  - Your template definition is NOT runnable code
  - Code is *only* generated if you use your template

Function Templates

Template to **compare** two “things”:

```
#include <iostream>
#include <string>

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>   // <...> can also be written <class T>
int compare(const T& value1, const T& value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}

int main(int argc, char **argv) {
  std::string h("hello"), w("world");
  std::cout << compare<int>(10, 20) << std::endl;
  std::cout << compare<std::string>(h, w) << std::endl;
  std::cout << compare<double>(50.5, 50.6) << std::endl;
  return EXIT_SUCCESS;
}
```

Only uses operator < to minimize requirements on T

Explicit type argument

functiontemplate.cc

Compiler Inference

Same thing, but letting the compiler infer the types:

```
#include <iostream>
#include <string>

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>
int compare(const T& value1, const T& value2) {
  if (value1 < value2) return -1;
  if (value2 < value1) return  1;
  return 0;
}

int main(int argc, char **argv) {
  std::string h("hello"), w("world");
  std::cout << compare(10, 20) << std::endl; // ok, infers int
  std::cout << compare(h, w) << std::endl;   // ok, infers string
  std::cout << compare("Hello", "World") << std::endl;  // hm…
  return EXIT_SUCCESS;
}
```

Infers char* - does address integer comparison

functiontemplate_infer.cc

Template Non-types

You can use non-types (constant values) in a template:

```
#include <iostream>
#include <string>

// return pointer to new N-element heap array filled with val
// (not entirely realistic, but shows what's possible)
template <typename T, int N>
T* valarray(const T& val) {
  T* a = new T[N];
  for (int i = 0; i < N; ++i)
    a[i] = val;
  return a;
}

int main(int argc, char **argv) {
  int* ip = valarray<int, 10>(17);
  string* sp = valarray<string, 17>("hello");
  ...
}
```

Fixed type template parameter

Use comma separated list to specify template arguments

valtemplate.cc

What's Going On?

The compiler doesn't generate any code when it sees the template function

- It doesn't know what code to generate yet, since it doesn't know what types are involved

When the compiler sees the function being used, then it understands what types are involved

- It generates the **instantiation** of the template and compiles it (kind of like macro expansion)
  - The compiler generates template instantiations for *each* type used as a template parameter

Class Templates

Templates are useful for classes as well

- (In fact, that was one of the main motivations for templates!)

Imagine we want a class that holds a pair of things that we can:

- Set the value of the first thing
- Set the value of the second thing
- Get the value of the first thing
- Get the value of the second thing
- Swap the values of the things
- Print the pair of things

Pair Class Definition

Pair.h

```
#ifndef PAIR_H_
#define PAIR_H_

template <typename Thing> class Pair {
 public:
  Pair() { };

  Thing get_first() const { return first_; }
  Thing get_second() const { return second_; }
  void  set_first(Thing& copyme);
  void  set_second(Thing& copyme);
  void  Swap();

 private:
  Thing first_, second_;
};

#include "Pair.cc"
#endif  // PAIR_H_
```

Template parameters for class definition

Could be objects, could be primitives

Pair Function Definitions

Pair.cc

```
template <typename Thing>
void Pair<Thing>::set_first(Thing& copyme) {
  first_ = copyme;
}
template <typename Thing>
void Pair<Thing>::set_second(Thing& copyme) {
  second_ = copyme;
}
template <typename Thing>
void Pair<Thing>::Swap() {
  Thing tmp = first_;
  first_ = second_;
  second_ = tmp;
}
template <typename T>
std::ostream &operator<<(std::ostream& out, const Pair<T>& p) {
  return out << "Pair(" << p.get_first() << ", "
             << p.get_second() << ")";
}
```

Definition of member function of template class

member of template class

Non member function to print out data in template class

Using Pair

usepair.cc

```
#include <iostream>
#include <string>

#include "Pair.h"

int main(int argc, char** argv) {
  Pair<std::string> ps;
  std::string x("foo"), y("bar");

  ps.set_first(x);      // ("foo", "")
  ps.set_second(y);     // ("foo", "bar")
  ps.Swap();            // ("bar", "foo")
  std::cout << ps << std::endl;

  return EXIT_SUCCESS;
}
```

Invokes default ctor, which default constructs members ("", "")