

Additional Topics (templates, runtime vs compile time polymorphism, type conversion/casting)

Templates:

(just an intro, will do more examples later on)

The purpose of templates is to allow you to write generic programs. Instead of specifying what type you are working with (like *int* or *char*), we just write a generic function (or class) to allow for different types to be used.

```
computer$ g++ -std=c++11 practice.cpp
computer$ ./a.out

The sum of integer values : 300

The sum of long values : 34

Two strings : hello world!
New value in c1: 6
```

Example 1: (template function)

```
#include<iostream>
```

```
class Cat{
public:
    int num=3;
```

```
    Cat operator+(Cat c) //overloaded operator
    {
        Cat c1;
        int value;
        value=this->num+c.num;
        c1.num=value;
        return c1;
    }
```

```
};
```

//Template function. We can pass in different types for the function (instead of specifying it for one type)
//Think of the T as a placeholder meaning any type (int or string for example)

```
template <class T>
```

```
T plus_sign(T n1, T n2) //takes two parameters of type T and returns something of type T (some type)
```

```
{
    T ret;

    ret = n1 + n2;
```

```
    return ret;
}
```

```
int main()
```

```
{
    int num1=100,num2=200,num3;
```

```
long long1=13,long2=21,long3;  
std::string s1="hello", s2=" world!", s3;  
Cat c1, c2, new_cat; //each cat start with the num 3
```

//Notice we can use the template function with any of the types:

```
num3 = plus_sign(num1,num2);  
std::cout<<"\nThe sum of integer values : "<<num3<<std::endl;
```

```
long3 = plus_sign(long1,long2);  
std::cout<<"\nThe sum of long values : "<<long3<<std::endl;
```

```
s3=plus_sign(s1,s2);  
std::cout<<"\nTwo strings : "<<s3<<std::endl;
```

```
new_cat=plus_sign(c1, c2);
```

std::cout<<"New value in c1: "<<new_cat.num<<std::endl; //remember all Cat objects start with the num 3 and we defined + to be an overloaded operator that adds two Cat objects together (meaning adding the nums together), so it is 6

```
}
```

Example 2: (template class)

```
computer$ g++ -std=c++11 practice.cpp  
computer$ ./a.out  
100  
a
```

```
#include <iostream>  
using namespace std;
```

```
template <class T>  
class Temp_example {  
    T e1, e2; //type not specified  
public:  
    Temp_example (T first, T second) //type not specified  
    {  
        e1=first;  
        e2=second;  
    }  
}
```

//template <class T>

```
T larger ()  
{  
    T retval;  
    retval = e1>e2? e1 : e2; //is e1 greater than e2 (T/F)? Return e1 if true, e2 if false  
    return retval;
```

```
}  
};
```

//syntax to define function outside of class:

```
// template <class T>  
// T Temp_example<T>::larger ()  
// {  
//   T retval;  
//   retval = a>b? a : b;  
//   return retval;  
// }
```

```
int main (int argc, char **argv) {  
    Temp_example <int> example1 (100, 75); //100 is larger  
  
    Temp_example <char> example2('A','a'); //the letter a is larger (check ASCII-97), A is less (check ASCII-55)  
  
    cout << example1.larger()<<endl;  
    cout<< example2.larger()<<endl;  
    return 0;  
}
```

Both of the following will you get this output:

```
computer$ ./a.out  
~~~Smoothie info: Strawberry $4.99  
~~~Brownie info: Cheesecake Brownie $9.99
```

No template function:

```
#include <iostream>  
using namespace std;
```

```
class Smoothie  
{  
    private:  
        string flavor;  
        double price;  
    public:  
        Smoothie(double price, string flavor) : flavor(flavor), price(price) {}  
        friend ostream & operator<<(ostream& os, Smoothie& s); //friend, not a member of the class  
};
```

```
class Brownie  
{  
    private:  
        string flavor;  
        double price;  
    public:  
        Brownie(double price, string flavor) : flavor(flavor), price(price) {}  
};
```

```

    friend ostream & operator<<(ostream& os, Brownie& b1); //friend, not a member of the class
};

ostream & operator<<(ostream& os, Smoothie& s)
{
    os <<"~~~Smoothie info: "<< s.flavor <<" $"<<s.price;
    return os;
}

ostream & operator<<(ostream& os, Brownie& b1)
{
    os <<"~~~Brownie info: "<< b1.flavor <<" $"<<b1.price;
    return os;
}

int main(int argc, char **argv)
{
    Smoothie s1(4.99, "Strawberry");
    cout<<s1<<endl;

    Brownie b1(9.99, "Cheesecake Brownie");
    cout<<b1<<endl;
    return 0;
}

```

With a template function:

```

#include <iostream>
using namespace std;

class Smoothie
{
private:
    string flavor;
    double price;
public:
    Smoothie(double price, string flavor) : flavor(flavor), price(price) {}
    friend ostream & operator<<(ostream& os, Smoothie& s); //friend, not a member
};

class Brownie
{
private:
    string flavor;
    double price;
public:
    Brownie(double price, string flavor) : flavor(flavor), price(price) {}
    friend ostream & operator<<(ostream& os, Brownie& b1); //friend, not a member
}

```

```

};

ostream & operator<<(ostream& os, Smoothie& s)
{
    os <<"~~~Smoothie info: "<< s.flavor <<" $"<<s.price;
    return os;
}

ostream & operator<<(ostream& os, Brownie& b1)
{
    os <<"~~~Brownie info: "<< b1.flavor <<" $"<<b1.price;
    return os;
}

namespace template_example{
    template <class T> //template function-can print either smoothie or brownie
    void print_info(T e1)
    {
        cout<<e1<<endl;
    }
}

int main(int argc, char **argv)
{
    Smoothie s1(4.99, "Strawberry");
    Brownie b1(9.99, "Cheesecake Brownie");

    template_example::print_info(s1);
    template_example::print_info(b1);
}

```

Runtime vs Compile Time Polymorphism:

When making a decision about which function to call based on a specific situation, do we decide at compile time or runtime?

Overloaded functions are examples of compile time polymorphism

- The function to be used is decided when compiling the program
- This decision is based on parameters used (remember they are diff)
- Overloaded operators are also examples of compile time polymorphism
- You may also hear this term referred to as static polymorphism or early binding

Overridden functions are examples of runtime polymorphism

- We decide which function we are using during runtime (when the program is running)
- You may also hear this term referred to as dynamic polymorphism or late binding

```

computer$ g++ practice.cpp
computer$ ./a.out
~~~Doing work stuff!!
I'm studying!
Are you fun or boring?
fun
~~~LOOK AT ME DANCE!!! I'M SO FUN!!! SHAKE SHAKE SHAKE!!!

computer$ ./a.out
~~~Doing work stuff!!
I'm studying!
Are you fun or boring?
boring
Dancing is not my thing. //since I picked boring this time, the Boring_person function
is called. This was decided during the run of the program.

```

```
#include <iostream>
```

```
class Person{
```

```
public:
```

```
void work() //overloaded function
{
    std::cout<<"~~~Doing work stuff!!"<<std::endl;
}
```

```
void work(std::string action) //overloaded function
{
    std::cout<<"I'm "<<action<<"ing!"<<std::endl;
}
```

```
virtual void dance() //overridden function in derived classes
{
    std::cout<<"Generic dance."<<std::endl;
}
```

```
};
```

```
class Fun_person: public Person{
```

```
virtual void dance() //overridden function
{
    std::cout<<"~~~LOOK AT ME DANCE!!! I'M SO FUN!!! SHAKE SHAKE SHAKE!!!"<<std::endl;
}
```

```
};
```

```
class Boring_person:public Person{
```

```
virtual void dance() //overridden function
{
    std::cout<<"Dancing is not my thing."<<std::endl;
}
```

```
};
```

```

int main(int argc, char **argv)
{
    Person p1;
    std::string answer;

    p1.work(); //when the program is compiled, it knows at that time which function to call based on the number
of arguments given
    p1.work("study");

    Person *p2;

    std::cout<<"Are you fun or boring?"<<std::endl;
    std::cin>>answer;

    if(answer=="fun")
    {
        p2=new Fun_person();
    }

    else
    {
        p2=new Boring_person();
    }

    p2->dance(); //we know which dance() function to call only during the run of the program. In this case, we had
to wait for user input to know which one to run. Don't forget to delete your pointers.

}

```

Type Conversion:

There are two types of types conversion we will learn about:

- Implicit conversion
- Explicit conversion

Implicit conversion:

- The conversion happens behind the scenes (we don't notice or have anything to do with the actual conversion taking place, just the outcome)
- The compiler does it for us
- You may also see it referred to as *coercion*

Example:

(note: this is just a warning when compiling, meaning we can still run the program)

```

computer$ g++ practice.cpp
practice.cpp:49:8: warning: implicit conversion from 'double' to 'int' changes
      value from 3.2 to 3 [-Wliteral-conversion]
      int s=3.2;
           ^~~
1 warning generated.

```

This warning could be useful since I am losing some information by it converting from a double type to an int type (losing the .2)

```
#include <iostream>
```

```
int main(int argc, char**argv)
{
    int s=3.2;
    std::cout<<s<<std::endl;
}
```

Explicit conversion:

- Explicit conversion is when the programmer explicitly directs the type conversion
- This is called casting

(notice the warning does not pop up now because the compiler is assuming the programmer knowingly changed types and is aware they are losing the .2 bit of info)

```
computer$ g++ practice.cpp
computer$ ./a.out
3
```

```
#include <iostream>
```

```
int main(int argc, char** argv)
{
    int s=(int)3.2; //using a cast to convert a float to an int-using (int) before the float. This is called a C-style cast
    std::cout<<s<<std::endl;
}
```

We will discuss two additional types of casts available to you in C++ (note there are more than these two):

- **Static casting**
- **Dynamic casting**

Static casting:

- This cast checks at compile time, not runtime (see dynamic casting below for comparison)

The following will compile with no error:

```
#include <iostream>
```

```
class Host{
```

```
};
```

```
class Late_night_host:public Host{
```

```
};
```

```
int main(int argc, char **argv)
{
```



```
Host* h1;
int number=4;
```

```
h1=(Host*)&number); /*I'm casting an int to a Host-I shouldn't do this, but I don't get an error*/
}
```

The following (using a static_cast) does give an error when I try to compile:

```
computer$ g++ practice.cpp
practice.cpp:16:5: error: static_cast from 'int *' to 'Host *'
      is not allowed
      h1=static_cast<Host*>(&number);
          ^~~~~~
1 error generated.
```

```
#include <iostream>
```

```
class Host{
```

```
};
```

```
class Late_night_host:public Host{
```

```
};
```

```
int main(int argc, char **argv)
```

```
{
```

```
    Host* h1;
```

```
    int number=4;
```

```
    h1=static_cast<Host*>(&number); /*this will cause an error*/
```

```
    number2=static_cast<int>(number); //sidenote: notice I can also do it with types like int and float also
```

```
}
```

Dynamic casting:

- Checks at runtime time
- Good for polymorphism

Notice that the program catches the bad cast at runtime (if I changed from dynamic_cast to static_cast, this bad cast would not have been caught during runtime since static_cast does not do any checks during runtime-the program would just run. This can be dangerous later on in the program if we allow a cast to occur that shouldn't occur)

```
computer$ g++ practice.cpp
Computers-MacBook-Air:C++ computer$ ./a.out
Fail first.
```

```
#include <iostream>
```

```
class Student{
```

```
    virtual void action(){}
```

```
};
```

```
class Grad_student:public Student {};
```

```
int main(int argc, char **argv) {
    Student *s1 = new Student();
    Grad_student *b = dynamic_cast<Grad_student*>(s1); //fails...s1 is pointing at a student (new Student());
    NOT new Grad_student() ), so I can't cast to grad_student...dynamic_cast will return NULL to indicate this
    if (b == NULL)
    {
        std::cout << "Fail first.\n";
    }

    Student *s2 = new Grad_student();
    b = dynamic_cast<Grad_student *>(s2); // succeeds...s2 is actually pointing at a grad student, so we can cast
    to a grad student pointer
    if (b == NULL)
    {
        std::cout << "Fail second.\n";
    }

    return 0;
}
```

Note that the class I'm working with (the base class Student in this case should be polymorphic. I get the following error if I take out the following line in my Student class (a class that is considered polymorphic if it declares or inherits a virtual function):

```
virtual void action(){};
```

```
computer$ g++ practice.cpp
practice.cpp:11:23: error: 'Student' is not polymorphic
    Grad_student *b = dynamic_cast<Grad_student*>(s1);    //f...
                      ^~~~~~
practice.cpp:18:9: error: 'Student' is not polymorphic
    b = dynamic_cast<Grad_student *>(s2);    // succeeds.....
        ^~~~~~
2 errors generated.
```

When dealing with references, you can catch a bad cast when an exception is thrown (you can have a NULL pointer, but not a NULL reference):

```
computer$ g++ practice.cpp
computer$ ./a.out
Failed here: std::bad_cast
```

```
#include <iostream>
#include <exception>
```

```
class Student{
    virtual void action(){}
};
```

```
class Grad_student:public Student {};
```

```

int main(int argc, char **argv) {
    Student s_example;
    Student &s1 = s_example;

    try{
        Grad_student &b = dynamic_cast<Grad_student*>(s1);
    }

    catch(const std::bad_cast& e) { //bad_cast is a derived class from exception
        std::cout<<"Failed here: "<<e.what()<<std::endl;
    }
}

```

Using a virtual function vs dynamic casting:

Casting is useful when we can't add a virtual function to the base or have a function that only exists in the derived class:

```

computer$ g++ practice.cpp
computer$ ./a.out
Going to the branch...
Jumping!

```

```

#include <iostream>

```

```

class Bird{
public:
    virtual void go_home()=0;
};

```

```

class Lovebird:public Bird{

```

```

public:

    virtual void go_home()
    {
        std::cout<<"Going to the branch..."<<std::endl;
    }

```

```

    void jump()
    {
        std::cout<<"Jumping!"<<std::endl;
    }

};

```

```

class Cardinal:public Bird{

```

```

public:
    virtual void go_home()
    {
        std::cout<<"Going to the nest..."<<std::endl;
    }

```

```

}

};

int main(int argc, char **argv)
{
    Bird *b1=new Lovebird();

    b1->go_home(); //since go_home is virtual in the base class we can call the overridden function (using the base
pointer) like this....

    //b1->jump(); //but how can we do this? jump() is not in the base class (only the in the derived), so if I try to
call it, I will get an error

    Lovebird* l1=dynamic_cast<Lovebird*>(b1); /*I can cast it! Note I could also use a static cast here if I knew for
certain b1 would be a derived Lovebird, not Cardinal*/

    l1->jump(); /*now this works*/

}

```

Another example:

```

C++ computer$ ./a.out
How many sports items are you getting?
3
Is this a baseball or basketball?
baseball
Making a baseball pointer!

Is this a baseball or basketball?
basketball
Making a basketball pointer!

Is this a baseball or basketball?
baseball
Making a baseball pointer!

Hit ball with bat.
Homerun!!

Put ball in hoop.
Dribble!!

Hit ball with bat.
Homerun!!

```

```
#include <iostream>
```

```

#include <string>
#include <vector>
#include <exception> // for std::exception

using namespace std;

class Sports_item{

public:
    float price;
    void virtual action()=0;

};

class Basketball: public Sports_item{

public:
    void action()
    {
        cout<<"Put ball in hoop."<<endl;
    }

    //this function is specific to the Basketball class
    void basketball_action()
    {
        cout<<"Dribble!!\n"<<endl;
    }

};

class Baseball: public Sports_item{

public:
    void action()
    {
        cout<<"Hit ball with bat."<<endl;
    }

    //this function is specific to the baseball class
    void baseball_action()
    {
        cout<<"Homerun!!\n"<<endl;
    }

};

class Sports_bag{
    vector<Sports_item*> items; //you should free memory when done with this

public:

```

```

void add_sports_item()
{
    string answer;
    Sports_item* s1;
    cout<<"Is this a baseball or basketball?"<<endl;
    cin>>answer;

    if(answer=="baseball")
    {
        cout<<"Making a baseball pointer!\n"<<endl;
        s1=new Baseball();
    }

    else
    {
        cout<<"Making a basketball pointer!\n"<<endl;
        s1=new Basketball();
    }

    items.push_back(s1);
}

void show_items()
{
    for(int i=0;i<items.size();i++)
    {
        Baseball* b1=dynamic_cast<Baseball*>(items.at(i));
        if(b1==NULL) //not pointing at a baseball, so must be a basketball (dynamic cast will return null if it does
not successfully cast because the types don't match)
        {
            Basketball* b2=dynamic_cast<Basketball*>(items.at(i));
            b2->action();
            b2->basketball_action();
        }

        else //it is a baseball (meaning the dynamic cast did not return null and succeeded)
        {
            b1->action();
            b1->baseball_action();
        }
    }
};

int main(int argc, char**argv)
{
    Sports_bag s1;
    int answer, counter=0;
    cout<<"How many sports items are you getting?"<<endl;
    cin>>answer;

```

```

while(counter<answer)
{
    s1.add_sports_item();
    counter++;
}

s1.show_items();

}

```

Remember if we are dealing with references instead of pointers, we don't return a null pointer to indicate that our dynamic cast did not work out. We throw an exception of `bad_cast`:

```

computer$ ./a.out
How many sports items are you getting?
3
Is this a baseball or basketball?
baseball
Making a baseball reference!
Homerun!!

Is this a baseball or basketball?
basketball
Making a basketball reference!
std::bad_cast
Dribble!!

Is this a baseball or basketball?
basketball
Making a basketball reference!
std::bad_cast
Dribble!!

```

```

#include <iostream>
#include <string>
#include <vector>
#include <exception> // for std::exception

using namespace std;

class Sports_item{

public:
    float price;
    void virtual action()=0;

};

```

```

class Basketball: public Sports_item{

public:
    void action()
    {
        cout<<"Put ball in hoop."<<endl;
    }

    //this function is specific to the Basketball class
    void basketball_action()
    {
        cout<<"Dribble!!\n"<<endl;
    }

};

class Baseball: public Sports_item{

public:
    void action()
    {
        cout<<"Hit ball with bat."<<endl;
    }

    //this function is specific to the baseball class
    void baseball_action()
    {
        cout<<"Homerun!!\n"<<endl;
    }

};

namespace randomstuff{

    void do_stuff(Sports_item& input)
    {
        //try to dynamically cast reference-you will get an exception if it is not successfully done
        try{
            Baseball &b1=dynamic_cast<Baseball&>(input);
            b1.baseball_action();
        }

        catch(std::bad_cast& ex) //bad cast is a derived type of exception (see below)
        {
            cout<<ex.what()<<endl;

            try{
                Basketball &b1=dynamic_cast<Basketball&>(input);
                b1.basketball_action();
            }

```



```
    catch(std::bad_cast&ex) //we don't need std:: because we have using namespace std, but just putting it to show you when you don't use using namespace
```

```
    {  
        //this will not execute because we only have two options in this program, but just showing you the syntax  
    }  
}  
}
```

```
int main(int argc, char**argv)  
{
```

```
    int answer, counter=0;  
    string answer1;  
    cout<<"How many sports items are you getting?"<<endl;  
    cin>>answer;
```










```
    while(counter<answer)  
    {  
        cout<<"Is this a baseball or basketball?"<<endl;  
        cin>>answer1;
```

```
        if(answer1=="baseball")  
        {  
            cout<<"Making a baseball reference!"<<endl;  
            Baseball b;  
            Sports_item& s=b;  
            randomstuff::do_stuff(s);  
        }
```

```
    else  
    {  
        cout<<"Making a basketball reference!"<<endl;  
        Basketball b;  
        Sports_item& s=b;  
        randomstuff::do_stuff(s);  
    }  
    counter++;  
}  
}
```

Remember *bad_cast* is a derived class from the exception class (base):

Derived types

Sr.No.	Derived types	Definition
1	bad_alloc 	This exception thrown on failure allocating memory
2	bad_cast 	This exception thrown on failure to dynamic cast
3	bad_exception 	This an exception thrown by unexpected handler
4	bad_function_call 	This exception thrown on bad call
5	bad_typeid 	This exception thrown on typeid of null pointer
6	bad_weak_ptr 	It is a bad weak pointer
7	ios_base::failure 	It is a base class for stream exceptions
8	logic_error 	It is a logic error exception
9	runtime_error 	It is a runtime error exception

https://www.tutorialspoint.com/cpp_standard_library/exception.htm

Problem 1:

Given the following main and sample run, complete the program. 4.99 is the price of a Smoothie object and *Strawberry* is the flavor. Both should be private member variables.

```
int main(int argc, char **argv)
{
    Smoothie s1(4.99, "Strawberry");
    cout<<s1<<endl;
    return 0;
}
```

Sample run:

```
computer$ ./a.out
Smoothie info:Strawberry 4.99
```

```
#include <iostream>
```

```
using namespace std;
```

```
class Smoothie {
private:
    string flavor;
    double price;
public:
```

```

    Smoothie(double price, string flavor) : flavor(flavor), price(price)
    {}
    friend ostream & operator<<(ostream& os, Smoothie& s);
};

ostream & operator<<(ostream& os, Smoothie& s)
{
    os <<"~~~Smoothie info: "<< s.flavor <<" "<<s.price;
    return os;
}

int main(int argc, char **argv)
{
    Smoothie s1(4.99, "Strawberry");
    cout<<s1<<endl;
    return 0;
}

```

Problem 2:

Show the countdown for a hide and seek game given the sample run. Each number should be printed out a second after the previous one.

Sample run:

```

computer$ ./a.out
How many seconds do you need to hide?
4
Ok:
4
3
2
1
Ready or not, here I come!

```

```

#include <iostream>
#include <thread>
#include <chrono>

using namespace std;

int main(int argc, char **argv)
{
    int answer;
    std::cout<<"How many seconds do you need to hide?"<<std::endl;
    cin>>answer;
}

```

```

std::cout << "Ok:\n";
for (int i=answer; i>0; i--) {
    std::cout << i << std::endl;
    std::this_thread::sleep_for (std::chrono::seconds(1));
}
std::cout << "Ready or not, here I come!\n";

return 0;
}

```

Problem 3:

Create the UML Class diagram for the following class (on board).

```

template <class T>
class Temp_example {
    T e1, e2; //type not specified
public:
    Temp_example (T first, T second) //type not specified
    {
        e1=first;
        e2=second;
    }

    //template <class T>
    T larger ()
    {
        T retval;
        retval = e1>e2? e1 : e2; //is e1 greater than e2 (T/F)? Return e1 if true, e2 if false
        return retval;
    }
};

```

Problem 4:

Create the UML Class diagrams for the following prompt (board).

ABC Airport allows two types of luggage: carry-on and check-in. All check-in bags must be under 50 pounds and all carry-on bags must be under 10 pounds. All bags have the functionality of packing items-if items are packed or not it should be indicated.