6.12 — Member selection with pointers and references

BY ALEX ON JULY 17TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

It is common to have either a pointer or a reference to a struct (or class). As you learned previously, you can select the member of a struct using the **member selection operator (.)**:

```
struct Person
{
    int age;
    double weight;
};
Person person;

// Member selection using actual struct variable
person.age = 5;
```

This syntax also works for references:

```
struct Person
{
    int age;
    double weight;
};
Person person; // define a person

// Member selection using reference to struct
Person &ref = person;
ref.age = 5;
```

However, with a pointer, you need to dereference the pointer first:

```
struct Person
{
    int age;
    double weight;
};
Person person;

// Member selection using pointer to struct
Person *ptr = &person;
(*ptr).age= 5;
```

Note that the pointer dereference must be enclosed in parentheses, because the member selection operator has a higher precedence than the dereference operator.

Because the syntax for access to structs and class members through a pointer is awkward, C++ offers a second member selection operator (->) for doing member selection from pointers. The following two lines are equivalent:

```
(*ptr).age  = 5;
ptr->age = 5;
```

This is not only easier to type, but is also much less prone to error because the dereference is implicitly done for you, so there are no precedence issues to worry about. Consequently, when doing member access through a pointer, always use the -> operator instead of the . operator.

Rule: When using a pointer to access the value of a member, use operator-> instead of operator. (the . operator)

6.12a -- For-each loops

Index

6.11a -- References and const

C++ TUTORIAL | PRINT THIS POST

38 comments to 6.12 — Member selection with pointers and references

cdecde57
June 5, 2019 at 9:41 am · Reply

So in the last unit, you mentioned to use references when possible instead of pointers. I find it easier to do the following.

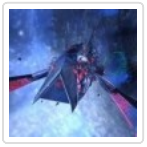
```
struct Player
{
std::string name;
int health;
int damage;
};


int main()
{
Player bob{"Bobby", 100, 10}; // Made a structure
std::string &uN{bob.name}; // uN means "userName"
uN = "Name Change";
}
```

Then to use pointers and have to worry about possible errors or things like that. Also becuase since it is a reference isn't it more memory efficient?

Anyways I am just wondering what to prefer, pointers or references or what do you prefer? I mean I will still learn them both the best I can and obviously, they are different and pointers can do many things references cant like dynamically allocating memory or like scanning words or cool things like that, but when it doesn't necessarily matter which do I prefer?

cdecde57
June 5, 2019 at 12:29 pm · Reply

Nvm I think I got a good idea on which to prefer. lol



potterman28wxcv
August 15, 2019 at 9:58 am · Reply

As a programmer you shouldn't care how memory efficient what you do is, unless you have a good reason to. That's called over-optimizing your code. You should privilege making your code readable and maintainable rather than optimized ; and if you later notice some inefficiencies harming the execution of your code, only then should you optimize it. I'm not expert on C++ compilers but I would not be surprised if references behave exactly like pointers when it comes down to generating machine code on some cases.

I guess the preferred usage of references compared to pointers is that references give more guarantees than pointers. You won't have any dangling pointer or memory leak with references. This comes at the cost of functionality - you can do less things with references than you can with pointers.

A few remarks on your code, first of all I would really indent it. I know it's just a tiny example, but if you start not indenting your code in small examples, this opens the door to being lazy on your bigger codes. And the lack of indentation can make a code very unreadable.

Also I would not do the following (but it's a matter of style):

```
std::string &uN{bob.name}; // uN means "userName"
```

If you feel like you need to express in a comment what your variable means, then your variable doesn't have a good name. Just call your variable `username`. Or `name` if you want something short.

Alex
August 16, 2019 at 9:04 pm · Reply

Just a quick note -- dangling references are a real thing. For example, a function that returns a reference to a local variable will pass back a dangling reference to the caller.

TPLDanino
February 17, 2019 at 8:21 pm · Reply

These tutorials are giving me such an incredible proper foundation for understanding what I write and why. I'd learnt bits and pieces of C++ before, but now I really feel like I'm writing good-quality code that conforms to best practices, is efficient and safe, and most of all that I fully understand and have control over!! For example, I finally understand the nuances and best uses for pointers, references, and their syntax, which always held me back before, hence why I'm writing this comment here.

Again, I can't express how grateful I am for these well-written and useful these tutorials are. Thanks so much Alex and co!!!!

Alex

February 18, 2019 at 8:12 pm · Reply

You're welcome. Thanks for visiting!

Sélim
December 8, 2018 at 3:43 am · Reply

Hi, while doing the exercize 2 of the quiz chapter 6, I noticed that this code won't compile

```
void GetStudentList(Student *listStudent, int lenght)
{
    for(int i = 0;i<lenght;i++)
    {
        std::cin >> listStudent[i]->name;
        std::cin >> listStudent[i]->grade;
    }
}
```

Apparently using the -> operator is a mistake and I just have to change the operator to the dot operator and it works as expected.
But could you explain while the -> operator is a mistake here since listStudent is a pointer ? IS it because i'm indirectly dereferencing the array with the
subscript operator [] ? Thanks in advance

nascardriver
December 8, 2018 at 5:19 am · Reply

```
listStudent
```

is a Student*

```
listStudent[i]
```

is a Student

To avoid confusion, I suggest using

```
void GetStudentList(Student listStudent[], int lenght)
```

for array parameters.

Sam
September 5, 2018 at 8:01 am · Reply

There's a slight typo, "parenthesis" should be "parentheses".

Alex
September 6, 2018 at 2:41 pm · Reply

Fixed! Thanks!

Nigel Booth
August 14, 2018 at 6:33 am · Reply

Hi,

Using the following code:

```
// TeamStructRef.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <iostream>
#include <string> //for string
#include <cstdlib>
#include <cstdio>



struct teamScores //a structure that may be useful someday
{
    std::string teamName{ " " };
    int goalsScored{ 0 };
    int totalPoints{ 0 };
};

void showTeam(teamScores &team) //team passed by reference so no expensive copying
{
    std::cout << "\nTeam: " << team.teamName << " , Goals scored: " << team.goalsScored <<
}

int main()
{

    int goals{ 0 };
    int length{ 0 };

    std::cout << "How many team scores to enter? ";
    std::cin >> length;

    for (int i{ 0 }; i < length; ++i)
    {
        std::cout << "Enter team name: ";
        std::string team;
        std::getline(std::cin, team);
        std::cout << "Enter goals scored: ";
        std::cin >> goals;
        int points{ goals * 4 };
        teamScores NF{ team, goals, points };
        showTeam(NF);
    }

    return 0;
}
```

it builds fine and runs but won't let me input the team name. Am I missing something? I used std::getline because some tams have names with white spaces in and std::cin stops at white space.

nascardriver
August 14, 2018 at 6:36 am · Reply

Hi Nigel!

@std::cin::operator>> leaves a '\n' in the input stream, @std::getline only reads up to the next '\n'. You need to @std::cin.ignore after using @std::cin::operator>>

Nigel Booth
August 14, 2018 at 6:50 am · Reply

Oops! Of course it does. The following is better and works correctly:

```cpp
// TeamStructRef.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <iostream>
#include <string> //for string
#include <cstdlib>
#include <cstdio>



struct teamScores //a structure that may be useful someday
{
    std::string teamName{ " " };
    int goalsScored{ 0 };
    int totalPoints{ 0 };
};

void showTeam(teamScores &team) //team passed by reference so no expensive copyin
{
    std::cout << "\nTeam: " << team.teamName << " , Goals scored: " << team.goals
}

int main()
{

    int goals{ 0 };
    int length{ 0 };

    std::cout << "How many team scores to enter? ";
    std::cin >> length;


    for (int i{ 0 }; i < length; ++i)
    {
        std::cin.ignore(32767, '\n');
        std::cout << "Enter team name: ";
        std::string team;
        std::getline(std::cin, team);
        std::cout << "Enter goals scored: ";
        std::cin >> goals;
        int points{ goals * 4 };
        teamScores NF{ team, goals, points };
        showTeam(NF);
    }

    return 0;
}
```

Thank you

nascardriver
August 14, 2018 at 6:54 am · Reply

* Line 19: Should be const reference
* Line 38: You know
* Line 36: Use @std::numeric_limits<std::streamsize>::max(). See the documentation of @std::cin.ignore
* You're using the same name style for types, functions, and variable. This will get confusing.

Rob G.
August 12, 2016 at 3:06 pm · Reply

Ok, thx for putting this together for me: using (.) operator and (->) selection operator on the object...

```
#include<iostream>

class Foo
{
private:
public:
    int a = 99;
    int b = 105;
    int c = 1001;
    void sing_it(){std::cout<<"Singing in the rain. Arrow op."<<std::endl;};
    void sing_it_again(){std::cout<<"Singing in the rain. Dot op."<<std::endl;};
};

int main()
{
    Foo obj;
    Foo &ref = obj;
    Foo * ptr_obj = &obj;

    std::cout<<ref.b<<std::endl;
    ptr_obj->sing_it_again(); //arrow op
    obj.sing_it_again();
    std::cout<<(*ptr_obj).a; //dot op
    std::cout<<obj.a; //same as (*ptr_obj).ae
    std::cout<<obj.c<<std::endl<<std::endl; //better dot op example
    ptr_obj->sing_it(); //arrow op
    obj.sing_it_again();

    return 0;
}
```

Rob G.
August 11, 2016 at 9:45 am · Reply

Gosh hate to post yet another question on this one chapter but, here goes: why is C++ configured so that we use aliases often. Why do we use aliases instead of original values? What function does the alias serve?

Alex
August 12, 2016 at 5:14 pm · Reply

If by "aliases" you mean references, we use (const) references wherever we can because they prevent values from being needlessly copied (especially when transferring an argument to a function parameter).

Rob G.
August 13, 2016 at 8:24 am · Reply

I understand that -- you have some really good teaching pts. there. What tripped me up was the insistence of some to call references aliases. I need to disambiguate: e.g.

"When a reference is created, you must tell it which variable it will become an alias for."

"A reference variable is an alias, that is, another name for an already existing variable."

These are from "teaching sites".

So naturally I thought maybe I missed a teaching point.

Matt
March 29, 2018 at 10:39 am · Reply

Rob,
I'm just a fellow student, but hopefully this helps to clarify...

```cpp
#include "stdafx.h"
#include <iostream>
#include <string> // for std::string

struct Employee
{
    int id{ 0 };
    int age{ 0 };
    std::string name{ "" };
};

// joe is passed by value, a copy of the entire
// struct is made for this function (not ideal)
void printByValue(Employee employee)
{
    std::cout << "Name: " << employee.name << ", Age: "
        << employee.age << ", ID: " << employee.id << '\n';
}

// but because this function takes a reference parameter
// joe is automatically converted to a reference, so
// no copy of the struct is made (even though we tried passing by value)
void printByRef(Employee &employee)
{
    std::cout << "Name: " << employee.name << ", Age: "
        << employee.age << ", ID: " << employee.id << '\n';
}

int main()
{
    Employee joe{ 127, 35, "Joe" };

    printByValue(joe);
    printByRef(joe); // looks like we are passing by value...

    // Alex may correct me, but I haven't really found
    // any use for declaring a reference like this, because
    // as shown above, any function that takes a ref parameter
    // will create one on its own when the it is passed a "value."
    // So there really isn't much use in creating our own references
    // like this.
    Employee &ref = joe;

    return 0;
}
```

Rob G.
August 9, 2016 at 8:23 am · Reply

Hi Alex

regarding

```
Test &ref = test
```

is &ref an alias for object test?
Is &ref referencing the address of test?
for e.g. [code]A * ptr = new A()

```

```

, deleting is done by deleting the pointer? ptr
for Object object, object.a (etc.), how is it destroyed? Delete only works on the heap

Reviewing some here.

Alex
August 10, 2016 at 7:18 pm · Reply

In Test &ref = test, ref is an alias for object test.
For A *ptr = new A(), you delete this by simply calling "delete ptr". We'll talk about what happens when objects are deleted (e.g. destructors) in chapter 8. And yes, delete only works with dynamically allocated memory, which means it only works on the heap.

Raghu
May 30, 2016 at 8:34 am · Reply

Alex loving your tutorial! thank for everything

J3ANP3T3R
May 19, 2016 at 7:11 pm · Reply

so like ... pointers hold memory address only and to get the value of a memory address that the pointer is pointing to you add * before the pointer variable.

but can anyone explain why ptr->age = 5; ?

i mean it seems like ptr now is a reference to the value of the memory address. it has become what data type the memory address is holding. why is it not *ptr->age = 5; ?

Alex

May 21, 2016 at 10:21 am · Reply

ptr->age is a "shortcut" for (*ptr).age. Operator-> does an implicit dereference (for convenience).

J3ANP3T3R
May 21, 2016 at 4:15 pm · Reply

That makes perfect sense thanks

Maxwell
April 8, 2016 at 8:10 pm · Reply

Thank you for everything Alex. You have brought me from total noobiness to the point where I can output dynamically allocated 2d arrays of chars in the console. I have recommended your tutorials to a friend and look forward to reading more.

Alex
April 9, 2016 at 11:20 am · Reply

That's great! Thanks for visiting.

Shiva
April 7, 2016 at 12:32 pm · Reply

Alex,

Isn't the -> operator called the arrow operator? And the . operator called the dot operator? Since you didn't mention them I was wondering if they were colloquial names. Then, in the last sentence:

```
> ... always use operator-> instead of operator.
                                              ^ dot operator or full stop? :D
```

IMHO you should at least say "... always use the -> operator instead of the . operator." if you're not using the terminology. Put the symbol before 'operator'.

That last sentence also deserves a rule status, doesn't it? Repeat in italics! :)

Alex
April 7, 2016 at 2:40 pm · Reply

. is often called "dot" and -> is often called "arrow", but these are colloquial names. I've updated the article based on your other suggestions. Thanks!

Rob G.
January 26, 2016 at 7:18 am · Reply

Thank you Alex, clear as always.

Rob G.
January 8, 2016 at 9:51 am · Reply

Hi Alex I have 2 questions. In the code below dereference occurs as a step if the -> is not used. What does the dereferencing produce? Literally what value/object is a result of dereferencing. (full code at bottom)
1)

```
...
(*n_ptr).value_1 = 5;//What results from dereferencing? With a regular pointer sometimes it
cout<<(n_ptr)->value_1;    //"awkward" pointer to struct member.
...
```

I've lost my way. What does the & do in this line?
2)

```
...
Mem_Vars * n_ptr = &mem_vars;
...
```

Full code:

```
        Mem_Vars mem_vars;                             //preferred way - arrows (does dereferenc

    Mem_Vars * n_ptr = &mem_vars;                                   //point n_ptr to struct

    cout<<(n_ptr)->value_1<<endl;
    (n_ptr)->value_1 = 45;                         //access members with pointer
    cout<<(n_ptr)->value_1<<endl;

    (*n_ptr).value_1 = 5;                                           //access members with p
    cout<<(n_ptr)->value_1;                                         //undesirable way
```

Alex
January 9, 2016 at 3:59 pm · Reply

1) `(*n_ptr).value_1 = 5;` is the same as `n_ptr->value_1 = 5;`, so you're just assigning the value of 5 to the value_1 member of mem_vars.
2) In this context, & gets the address of mem_vars, so the pointer can point to it. Remember that pointers hold addresses.

Paulo Henrique
April 14, 2015 at 4:45 am · Reply

```
// Why a const pointer pointing to a const object is not allowed in c++? Something like tha

const int value = 2;
int *const constPointer = &value; // compilation error.


// Const reference can referencing a const object :

const int value = 2;
cont int &ref = value;  // works fine
```

Arduino_User
May 11, 2015 at 12:49 pm · Reply

This is covered in lesson 6.10.

Basically, you say that the pointer points to a constant address, NOT that it points to a constant value. The following lines are OK:

```
const int *constPointer = &value; // fine!
```

The value of the pointer cannot be changed; but the pointer can be assigned to another address. Or:

```
const int *const constPointer = &value; // Also fine!
```

Here, you cannot change the value of the pointer OR the address it points to.

zmotaghi67@gmail.com
December 24, 2014 at 4:38 am · Reply

i love you Alex.
every time i stuck and want to understand some thing deeply,i just come here... and i am sure i will get what i want.
you must write all those heavy books.
not these stupids called writers that just make things hazier!

dice3000
August 8, 2013 at 1:41 pm · Reply

Brain overflow. Need to take a rest :)

Vishal Singhal
November 29, 2010 at 8:39 am · Reply

What cannot be done using references in C++. Can we have array of references just like array of pointers?

Alex
August 21, 2015 at 2:43 pm · Reply

There are a few things you can't do with references (this is not an exhaustive list):

* You can't dynamically allocate memory using references.
* You can't have a null reference.
* You can't change where a reference points (they're implicitly const).
* You can't have a pointer to a reference (because a reference isn't considered a real object in C++).
* You can't have an array of references (because an array is essentially a pointer, and you can't have a pointer to a reference).

I'm sure there are others.